Network Device Discovery

DENYS KNERTSER
and
VICTOR TSARINENKO

KTH Information and
Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Network Device Discovery

Denys Knertser and Victor Tsarinenko

June 10, 2013

Abstract

Modern heterogeneous networks present a great challenge for network operators and engineers from a management and configuration perspective. The Tail-f Systems' Network Control System (NCS) is a network management framework that addresses these challenges. NCS offers centralized network configuration management functionality, along with providing options for extending the framework with additional features. The devices managed by NCS are stored in its Configuration Database (CDB). However, currently there is no mechanism for automatically adding network devices to the configuration of NCS, thus each device's management parameters have to be entered manually. The goal of this master's thesis project is to develop a software module for NCS that simplifies the process of initial NCS configuration by allowing NCS to automatically add network devices to the NCS CDB.

Apart from developing the software module for discovery, this project aims to summarize existing methods and to develop new methods for automated discovery of network devices with the main focus on differentiating between different types of devices. A credential-based device discovery method was developed and utilized to make advantage of known credentials to access devices, which allows for more precise discovery compared to some other existing methods. The selected methods were implemented as a component of NCS to provide device discovery functionality.

Another focus of this master's thesis project was the development of an approach to network topology discovery and its representation. The aim is to provide both a logical Internet Protocol (IP) network topology and a physical topology of device interconnections. The result is that we are able to automatically discover and store the topology representation as a data structure, and subsequently generate a visualization of the network topology.

Sammanfattning

Moderna heterogena nätverk utgör en stor utmaning för operatörer och ingenjörer att hantera och konfigurera. Tail-f Systems NCS produkt är ett ramverk för nätverks konfiguration som addresserar dessa utmaningar. NCS är ett centraliserat nätverks konfigurations verktyg. NCS är användbart som det är, men kan också byggas ut av användaren med ytterligare funktioner. De enheter som hanteras med NCS lagras i konfigurationsdatabasen (CDB). För närvarande finns det ingen automatiserad mekanism för att addera nätverksenheter till NCS, och varje enhets parametrar måste fyllas i manuellt. Detta examensarbetes mål är att utveckla en mjukvarumodul för NCS som förenklar NCS konfiguration genom att automatiskt lägga nätverksenheter till NCS CDB.

Förutom att utveckla programvara för enhetsidentifiering, syftar detta projekt till att sammanfatta befintliga metoder och utveckla nya metoder för automatiserad nätverksenhetsidentifiering med huvudfokus på att skilja mellan olika typer av enheter. En metod baserad på förkonfigurerade autenticeringsuppgifter har utvecklats och den används för att precist kunna identifiera olika typer av nätverkselement. De valda metoderna har implementerats som en optionell modul till NCS som erbjuder enhetsidentifieringsfunktionalitet.

Ytterligare ett fokus för detta examensarbete har varit att utveckla metoder för identifieraing av nätverkstopologin, och modeller för hur topologin ska representeras. Vi har syftat till att identifiera både den logiska IP nätverkstopologin (L3) och den fysiska topologin av sammankopplade enheter (L2). Den viktigaste uppgiften har varit att identifiera och lagra topologi representation som en datastruktur, och dessutom generera en visualisering av nätverkstopologin.

Acknowledgements

We would like to thank our industrial supervisors **Claes "Klacke" Wikström** and **Stefan Wallin** for all their help and invaluable feedback, as well as great discussions and providing us with a lot of ideas for the project. We would also like to thank everyone at Tail-f Systems for making us feel very welcome and for the friendly atmosphere, especially **Ulf Tennander**, **Jan Lindblad**, **Jane Carlgren**, **Christopher Williams**.

Professor **Gerald Q. "Chip" Maguire Jr.** has been a great academic supervisor who provided us with valuable information and continuous extensive feedback.

We are grateful to our families for their support which we felt despite the distance from home.

Special thanks to our friends for letting us from time to time forget about the project and making us feel happy.

Contents

List of Figures

List of Listings

List of Acronyms and Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
ASN.1	Abstract System Notation One
BER	Basic Encoding Rules
CDB	Configuration Database
CDP	Cisco Discovery Protocol
CGA	cryptographically generated address
CLI	Command Line Interface
DNS	Domain Name System
DU	Data unit
ECN	Explicit Congestion Notification
EDP	Extreme Discovery Protocol
FDP	Foundry Discovery Protocol
GPL	General Public License
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
ID	identifier
IDS	Intrusion Detection System

IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IOS	Internetwork Operating System
IP	Internet Protocol
IPS	Intrusion Prevention System
JunOS	Juniper Operating System
L2	Layer 2
L3	Layer 3
LAN	Local Area Network
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MIB	Management Information Base
MPLS	Multi-protocol Label Switching
NCS	Network Control System
NDP	Nortel Discovery Protocol
NED	Network Element Driver
NETCONF	Network Configuration Protocol
NMS	Network Management System
OID	Object Identifier
OS	operating system
OUI	Organizationally Unique Identifier
PDU	Protocol Data Unit
RPC	Remote Procedure Call
SEND	SEcure Network Discovery
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol
SSH	Secure Shell

SSL	Secure Sockets Layer
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLV	Type-Length-Value
TTL	Time to Live
UDP	User Datagram Protocol
VM	Virtual Machine
VPN	Virtual Private Network
WebUI	Web User Interface
WLAN	wireless local area network
XML	eXtended Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

Network management became a non trivial task, as networks grew and incorporated different types of devices. Manual network management of large scale networks is unfeasible due to the need for engineers specialized in different aspects and types of network devices and their management, limited time, need to define a strategy for configuration management, and the effort to track the configuration state of large number of different devices. These factors obviously increase the costs and effort required for network management. To overcome these difficulties Network Management Systems (NMSs) were developed.

An NMS is a tool for network operators and engineers. Such a tool enables centralized configuration management of many different network devices, consolidates the storage of device configuration and state information, pushes and pulls the configuration changes to and from the devices. Additionally, an NMS may provide visualizations of the network topology and state, provide an alert system with notifications, and automate network service deployment. However, designing and developing such a system is a non-trivial task. The reasons for this are the presence of different vendors in the network equipment market, the existence of different configuration interfaces to these network devices, and the lack of a single generic interface for configuration management. For instance, consider the network equipment vendors: Juniper and Cisco; both of these vendors have proprietary operating systems (OSs) on their devices and use proprietary configuration interfaces, which in turn requires an NMS to implement two completely different ways to manage these devices, and, ultimately, other configuration interfaces to manage other types of devices. Designing and developing such interfaces for every possible device type becomes impractical due to both the number of different devices and the evolution of their interfaces. A possible solution is to define a model for the device configuration structure which

serves as a structured representation of the device configuration and a protocol which provides an interface to access this model and to perform configuration changes.

Network Control System (NCS), an NMS developed by Tail-f Systems, is a network management system and framework based on the Network Configuration Protocol (NETCONF) and YANG (a protocol that provides a unified interface to different devices and a modeling language utilized by this protocol). The details of these are discussed in the next chapter. However, NCS needs to support not only NETCONF-enabled devices, but also devices that do not yet have NETCONF support, or will never have NETCONF support. NCS does this by enabling extensions to the system in the form of YANG defined models that mimic the configuration structure of devices. For instance, a Simple Network Management Protocol (SNMP) Management Information Base (MIB) can be defined in a YANG model and NCS can use SNMP to manipulate the configuration data according to the models.

1.1 Problem statement and project goals

NCS incorporates a vast number of features that enable automated and centralized device management. However, in order to further extend NCS's functionality and to minimize network management effort, a suitable device discovery mechanism is required. Currently, NCS supports manually adding devices to the system, thus the operator has to provide the address of the device and define its type. The device type in NCS specifies which (internal) interface NCS should use to communicate with the device (this internal interface will utilize NETCONF, SNMP, Cisco Command Line Interface (CLI), etc. to configure the device).

This thesis project will define a device discovery approach for use in an Internet Protocol (IP) network that will best suit NCS and enable NCS to provide automated device discovery functionality. However, device discovery is not a trivial task, since it requires not only determining the address of a device and whether it is accessible, but most importantly its type, i.e. whether the device is a Cisco Internetwork Operating System (IOS) device, Cisco Catalyst device, Juniper device, or some other kind of device. This is especially crucial for NCS, since the device type specifies which interface NCS will use to manage the device. Thus, the discovery module should be able to find NETCONF, CLI, and SNMP enabled devices and differentiate between them.

Device discovery, as defined in the context of this thesis, is a process of finding all the network devices in an address space specified by the network operator. The results of this network discovery process are represented as a list of devices defined in a YANG model (which includes device address and device type as primary attributes, as well as additional parameters, such as port numbers for control). The YANG model is then mapped to the NCS model and the devices are added to the NMS.

Another focus of this thesis is network topology discovery. The implementation of this task relies upon the SNMP and Cisco Discovery Protocol (CDP) protocols, thus, it is desirable for the network devices to support one or more of these protocols. Additional information about the network topology can be obtained from the devices' operational data, i.e. from the Media Access Control (MAC) address table of a network switch or looking at the status of each of device's interfaces.

In addition to the practical implementation of network device discovery, this project will also identify some of the common methods used for network device discovery and evaluate each of these methods. The results can serve as an input for further research into device and topology discovery, as well as for network management purposes. As network scanning and discovery has specific ethical concerns in some applications, a discussion of ethical aspects of network discovery and possible implications will also be presented.

1.2 Methodology

The initial requirement of this thesis project was to investigate methods that can be successfully utilized for network device and topology discovery, and as a result produce a set of methods that would suit the discovery component of NCS. Therefore, a significant part of this thesis project is devoted to the analysis and evaluation of current methods for device and topology discovery. This part of the project relies upon *qualitative* research methods. These qualitative research methods were used to develop an initial understating of the problem and to provide a foundation for further research and, in our case, implementation.

The actual implementation of the methods in the form of a discovery component for NCS is the main focus of this master's thesis project. The incremental approach to software development was adapted for this implementation. The development process mainly consists of three phases in the development of a single software component: planning, implementation, and testing. The planning

phase defined the overall functionality and approach used to develop the software component. The implementation phase was the actual implementation of the required functionality. Testing included the evaluation of the correct functionality of the component and error correction. An incremental approach was chosen due to its simplicity and cyclic nature. This approach is useful when requirements change or there is a need to introduce additional functionality. The development cycle for each component may be repeated until the components satisfy the requirements.

1.3 Restrictions and limitations

This thesis project was done at Tail-f Systems. All the source code developed during this project is the property of Tail-f Systems. The source code listings will be removed from the public version of this document, and will only be present in the internal version for the company. This document mainly discusses the ideas and approaches used during the project, describes the methods and their implementation used during the development phase of the project, provides suggestions for different aspects of discovery techniques, and discusses the associated ethical concerns of device discovery.

An important point to mention is the licensing restrictions of Nmap [1]. As part of this work makes use of the results produced by Nmap and relies somewhat on Nmap copyrighted data files (as Nmap relies on its **nmap-os-db** and **nmap-service-probes** for OS and service version detection), the use of this part of code may require further licensing before it can be used in a product.

1.4 Structure of the report

This report is organized as follows:

- Chapter 1 gives an introduction to the area of research, discusses the goals of the thesis, and the purpose of this thesis project;
- Chapter 2 discusses relevant background and summarizes the results of our literature study;
- Chapter 3 describes the implementation of the network device discovery component;

- Chapter 4 describes the implementation of the topology discovery component; and
- Chapter 5 concludes the thesis project, discusses the goals which have been reached, suggests future work, and describes the ethical considerations related to this project.

Chapter 2 gives an overview of the protocols and software products, as well as other information, needed to understand this thesis, or information that is otherwise relevant to this thesis project. Among other things, Chapter 2 describes NCS; as this software was the base for the practical part of this thesis project.

Chapter 2

Background

Several topics covered in our literature study are discussed in this chapter. One topic is network management protocols, as these are essential to understand any work in the area of network management. As one of the goals of this thesis project is to develop a device discovery component for NCS, an overview of NCS is presented in Section 2.2. Existing device discovery techniques are presented in Section 2.3. This chapter also mentions link-layer device discovery techniques and gives an overview of Link Layer Discovery Protocol (LLDP) and proprietary LLDP-like protocols which are essential for the topology discovery task. The final section gives an overview of the techniques for device discovery in an IPv6 network.

2.1 Network management protocols

Network management protocols define a means to access the configuration data of a managed device, as well as a means to change this data. This section will focus on the protocols which are essential in the context of working with NCS, namely SNMP, CLI, and NETCONF.

2.1.1 SNMP

SNMP [2] is a protocol for network management developed in late 1980s. The standard describes a protocol for exchanging management data between a management station and a managed device, the structure of a MIB, and a simple network management system architecture.

In its initial version SNMP assumed a centralized management system architecture which consists of a single management station and a number of managed nodes called SNMP agents. Later versions of SNMP added the ability of one management system to communicate with another, thus enabling a distributed network management system. Agents store data in the form of a tree whose leafs are variables called objects. The branches of this tree are numbered, so that each object can be uniquely represented by a sequence of branch numbers which leads from the root to the leaf of the tree. This protocol allows the management station to read a single variable or a set of variables from an SNMP agent, as well as to modify the value of a single writable variable or a set of writable variables. The standard also allows for asynchronous operations in which an agent sends a special type of message called an SNMP Notification (Trap) when a defined event happens (e.g. one of the links goes down).

MIBs define sets of management variables used by SNMP. A device can support one or several MIBs which can be standard or proprietary. From the point of view of the object tree, a MIB is a subtree which contains a set of leafs referenced by unique Object Identifiers (OIDs). A MIB consists of a set of modules. A module is a set of related management objects. A standard way to define a MIB module is to use the Structure of Management Information (SMI) language [3], a subset of Abstract System Notation One (ASN.1) language adapted for use with SNMP. SNMP utilizes ASN.1 to encode objects for transmission (specifically, Basic Encoding Rules (BER) of ASN.1 is used). SMI is intended to define a module's semantics as well as the syntax and semantics of the management objects and the syntax and semantics of the notifications [3].

A particular advantage of SNMP is the simplicity of the protocol and low complexity [4] of an agent implementation. This low complexity allows an SNMP agent to be embedded into even very limited resource devices. This feature made SNMP ubiquitous, so that today most network devices support SNMP agents.

However, SNMP has some disadvantages which prevent it from being the main protocol for network management today. Initially, SNMP had weak security, as SNMP was used primarily for reading information from the agents [4] and there was an assumption that only authorized and trusted users had access to the network. Even though the security model was improved in the third version of the protocol, there are still significant security issues, thus today SNMP is used primarily for fault management and performance management [5]. Another problem is the limited capabilities of the protocol. For example, SNMP is only able to read and write a single variable or a set of variables referenced

by sequences of numbers[1], thus making the task of developing a management system more complex. Although there are standardized sets of managed objects (e.g. MIB-II) which are recommended to be implemented, usually they do not provide enough flexibility, therefore most of the functionality of many devices is only available by using proprietary MIBs, which makes it difficult to operate heterogeneous networks. Generally, due to its variable-oriented nature SNMP is used to operate the devices by management software only, and SNMP is generally unsuitable for performing operations manually by humans.

2.1.2 CLI

The prevailing way to configure network devices today is using a CLI. This can be done locally (using wires to physically connect to a management port of the device), but more often is performed remotely (usually using Secure Shell (SSH) or telnet as a transport protocol). The advantage of this approach is that it offers maximum flexibility of configuration. However, each CLI is usually proprietary and even differs between different devices of the same manufacturer; this makes the task of managing a heterogeneous network non-trivial and requires multiple engineers who each specialize in different manufacturer's equipment. Also, although a CLI allows for automation of configuration activities to some extent, it is primarily intended to be used by humans, which makes developing management applications complex. However, in the context of this thesis we will speak about using the CLI interface offered by devices as if this was a protocol, as we will be sending CLI commands and parsing the results using software.

2.1.3 NETCONF

The Internet Engineering Task Force (IETF) NETCONF [6] is an extensible protocol designed to provide a generic interface to configure network devices. It is a fairly new standard and was developed to eliminate the previously discussed issues concerning the current means of configuration. Specifically, Schönwälder, Björklund, and Shafer [7] state that: "*The driving force behind NETCONF is the need for a programmatic cross-vendor interoperable interface to manipulate configuration state*".

NETCONF adopts a document-based approach to device configuration [5], this means that unlike SNMP and CLI it is possible to work with a device configuration

[1] It is also possible to transfer a number of consequent variables with SNMP GETBULK message by specifying the first OID in the sequence and the number of variables to retrieve

as a *structured document*, instead of working with a set of variables or a set of commands. NETCONF not only allows us to retrieve or submit configuration information to a device, but also to edit arbitrary parts of a configuration in a single transaction.

However, NETCONF by itself does not support message passing; thus it has to utilize some transport protocol. Relying upon standardized secure transport layers (SSH [8], Transport Layer Security (TLS) [9], and others [10, 11]) is a good practice from the security point of view, as these standardized protocols have been extensively studied by the research community. Additionally, it is easier to integrate credential management for NETCONF using an existing security management system, as opposed to using SNMP [5].

NETCONF makes use of eXtended Markup Language (XML) to represent the device configuration in the form of a document. The format of this document for a device is defined using a modeling language and is customizable, thus vendors can define their own configuration models depending on the devices and services they offer. Not only it is possible to define the structure of the device configuration data, but also the device's operational state data (which will also be accessible with NETCONF). While it is possible to specify the model using one of the standardized XML schema languages (such as XML Schema Definition (XSD)), the recommended language according to the IETF Network Configuration Working Group is YANG [12]. YANG [13] was specifically developed by the IETF NETCONF working group for this purpose.

The motivation behind YANG was to create an easily readable data modeling language which allows for a high degree of validation of a configuration datastore [7], which is crucial for automated network configuration management. YANG not only allows us to define the format for a device's configuration schema, but also allows us to define the necessary constraints which subsequently allow us to detect an invalid configuration, e.g. locking the "disabled" (or Cisco's "shutdown") option of a remote management interface allows us to make sure that the connection between the management station and the managed device will not be lost because of a mistake in the configuration.

2.2 NCS overview

NETCONF provides a generic interface to manipulate configuration data for network devices. However, in a large scale configuration management system we not only need a generic approach, we generally need a management

solution which allows network operators and engineers to minimize their efforts and to facilitate a consistent approach to configuration management. NCS provides a single configuration interface to a heterogeneous multi-vendor network infrastructure. NCS uses NETCONF as its primary configuration protocol and thus it directly supports NETCONF enabled devices. NCS is not just a network management system with specific built-in functionality, it is an extensible and scalable framework with a modular architecture which allows it to be very flexible and to integrate additional functionality. The modules in NCS are called packages. A package may be a YANG model that mimics a device configuration model, in this case such a package is called a Network Element Driver (NED). NEDs are used to define which configuration data sets of a device NCS can manage. Alternatively, a package may be a service model which defines how a common service, such as Multi-protocol Label Switching (MPLS), can be deployed on different devices. A package may also extend the functionality of NCS by adding custom functions and actions via NCS's Java Application Programming Interface (API).

NCS is comprised of three major parts: the service manager, the device manager, and the Configuration Database (CDB). Figure 1 provides a representation of NCS's logical architecture.

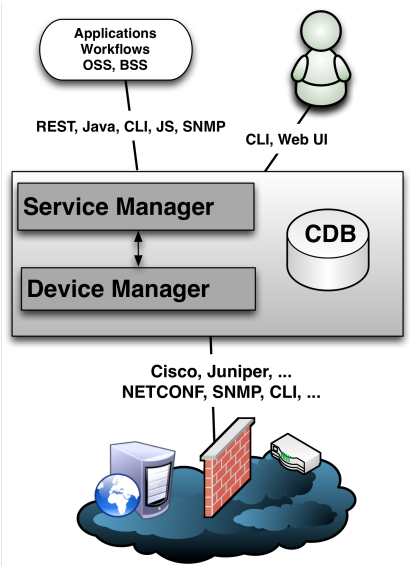

Figure 1: NCS logical architecture [14]
(appears here with permission from copyright owner)

The device manager functions as an interface to managed devices. It provides a means to add a new device to the CDB (a device may be added manually from a

device template or from an existing device configuration); to deploy configuration changes to managed devices in a fail-safe way, so that the changes can be applied simultaneously to any number of devices; to perform a configuration rollback; and to synchronize the NCS running the CDB with the actual configuration of the managed devices, which works in two ways (both to and from the devices).

The service manager defines higher layer functionality by acting as an interface to apply configuration changes to managed devices. It provides a means to model services (such as a Virtual Private Network (VPN) or MPLS) and defines the mapping of a service to managed devices via the device model.

The CDB is the core component of NCS and functions as a database for managed device configurations. The database also manages the relationship between service and device models. The data stored in the database may be defined as a configuration (configurable) or operational data (as determined by a **config** flag in the YANG model defined for a particular system component). Configuration data is defined as a configurable set of parameters which have read and write access from the NCS's user interface; this data defines configurable parameters both for NCS internal options and for remote device configuration management. Operational data is defined as a read-only (from the user interface perspective) informational data, which is useful for representing state data (such as routing tables or system uptime).

NCS provides several northbound (i.e. upward to higher layer applications) and southbound (i.e. downward to devices) interfaces. The northbound interfaces include CLI, Web User Interface (WebUI), and APIs. For instance, a NETCONF northbound interface can be used to provide access to the NCS to other applications. A Java API can be used to add custom applications and services to NCS in order to extend its functionality. A CLI not only provides a command line interface to manage NCS, but also supports scripting, which enables automation of tasks. The WebUI is a web based management interface which can be customized via packages. The southbound interfaces, on the other hand, provide the actual means for configuring the managed devices. The default protocol is NETCONF; therefore, if a device is NETCONF enabled and runs a NETCONF server, then NCS can automatically discover its capabilities and load the device model into its database. However, since NETCONF is not the prevalent configuration protocol at the moment, NCS provides other southbound interfaces as well. For Cisco style CLI devices, NCS provides a CLI model that can be used to configure such devices. NCS also has a number of commonly used MIBs as models to support devices that can be configured over SNMP. For other types of devices NCS uses NEDs, to represent a configuration model of a device defined in YANG.

2.3 Device discovery techniques

A number of approaches to discover network devices have been suggested by the research community. The method suggested by Schönwälder and Langendörfer [15] relies entirely on the Internet Control Message Protocol (ICMP) protocol, specifically ICMP echo request/reply, ICMP address mask request/reply, and ICMP port unreachable messages in order to extract a network's topology. However, this method will only work if ICMP traffic is not blocked (unfortunately, the majority of the systems do not respond to ICMP address mask messages, as they have to be explicitly configured as address mask agents, moreover this functionality is not mandatory to implement [16]). Additionally, ICMP provides no information about the device itself. Lin, Lai, and Chen [17] extend this approach by connecting to the SNMP interface of routers to fetch the configuration information, the routing tables, and contents of the Address Resolution Protocol (ARP) cache. This method may provide much better results. Finally, system information might also be read using SNMP, providing important information about the detected device's type. However, the SNMP interface may be closed or the credentials necessary to access it may be unknown.

Liu [18] and J. Wei-hua, L. Wei-hua, and Jun [19] suggest a more sophisticated approach to identifying the type of a network device, which includes analyzing various parameters collected during a communication session with this device. These parameters combined together form a so called fingerprint which identifies the OS of the target host. These parameters include a Time to Live (TTL) value, the default packet size, Initial Sequence Number, window size, Transmission Control Protocol (TCP)/IP flags, etc. Different OSs implement the TCP/IP stack differently, thus it is possible to guess the OS from these values. An analysis of the systems and fingerprints may be required to successfully and accurately perform OS prediction. One of the tools that can be used for OS fingerprinting is Nmap ("Network Mapper") [20]. Nmap contains a large database of fingerprints. It can also be extended by adding custom fingerprints.

A number of papers [21–23] focus on protocols for multimedia service discovery rather than device discovery. The goal of multimedia service discovery is also to find devices in the network, however the device which provides a service, i.e. a network printer or multimedia device, often wishes to be found and makes an effort to be reachable, for example by announcing its presence on the network. Although this kind of device can also be managed with NCS, the focus of this thesis project is on network infrastructure devices, such as routers and switches, which often do not announce their presence, or otherwise do it using standard network protocols, such as router advertisements in IPv6, therefore different

methods should be used to perform the search. Multimedia service discovery is outside the scope of this work, but may be investigated in the future in order to include this functionality into the network device discovery module that will be developed.

The main purpose for doing device discovery is not only to find active nodes on a network, but also be able to differentiate between the different types of nodes and to determine the OS or software platform running on a specific node. Device discovery can be performed passively or actively, where passive discovery occurs by passively listening to or "sniffing" network traffic, while active device discovery sends specifically crafted probe packets to target devices and analyzes the response(s). Passive discovery is a method that may require a long or predefined amount time (the period of time should be sufficient to produce a relatively reliable result) and is not very suitable for switched networks, hence it is not utilized frequently. The major focus in most cases is on active probing of network devices by sending connection requests, analyzing publicly available data from the devices, performing service banner grabbing, SNMP information gathering, and more in-depth network scanning.

SNMP is a prevalent protocol for gathering information about devices, including system and operational information. However, as mentioned earlier, due to the specific characteristics of this protocol it is mainly used for gathering statistical data and other information, rather than for remote configuration. Nevertheless, its prevalence makes SNMP one of the best bets for remote device discovery, considering that the required information, such as community strings (for SNMP version 1 and 2) or credentials (for SNMP version 3) are known. A specific OID that holds the identity of a hardware and software platform used by a system (i.e. **sysDescr** from MIB-II) can be queried and this information retrieved. Alternatively the vendor's identification of the system (i.e. **sysObjectID** from MIB-II) may be used to retrieve the platform identifier, however, the **sysDescr** object provides more extensive information. Another example of using SNMP is to gather information from the network devices such as the contents of a CDP neighbors list or an interface's configuration for network topology discovery. An example in Listing 1 demonstrates a request for a system description using SNMP version 3 and shows that information returned can be utilized to identify the type of a device.

```
denis@denis-laptop:~$ snmpwalk -v3 -l authPriv -u <username> -a <auth method> -A
    "<auth password>" -x <privacy method> -X "<privacy password>" 192.168.200.10
    1.3.6.1.2.1.1.1
iso.3.6.1.2.1.1.1.0 = STRING: "Linux savannah 3.2.0-29-generic-pae #46-Ubuntu SMP
     Fri Jul 27 17:25:43 UTC 2012 i686"
```

Listing 1: Obtaining system description string using SNMPv3

Due to SNMP's simplicity and prevalence it is also very useful for topology discovery. As JiaBin Yin [24] and Han Yan [25] point out, SNMP is the fundamental protocol for many existing topology discovery algorithms. Particularly useful information for topology discovery can be obtained form the routing tables in the devices. This information can help to map the network's topology. An example in Listing 2 illustrates an SNMP response to a request for routing information.

```
vits@y550 ~ $ snmpwalk -v 2c -c public 192.168.200.10 1.3.6.1.2.1.4.21
RFC1213-MIB::ipRouteDest.0.0.0.0 = IpAddress: 0.0.0.0
RFC1213-MIB::ipRouteDest.10.0.2.0 = IpAddress: 10.0.2.0
RFC1213-MIB::ipRouteDest.192.168.200.0 = IpAddress: 192.168.200.0
RFC1213-MIB::ipRouteIfIndex.0.0.0.0 = INTEGER: 2
RFC1213-MIB::ipRouteIfIndex.10.0.2.0 = INTEGER: 2
RFC1213-MIB::ipRouteIfIndex.192.168.200.0 = INTEGER: 3
RFC1213-MIB::ipRouteMetric1.0.0.0.0 = INTEGER: 1
RFC1213-MIB::ipRouteMetric1.10.0.2.0 = INTEGER: 0
RFC1213-MIB::ipRouteMetric1.192.168.200.0 = INTEGER: 0
RFC1213-MIB::ipRouteNextHop.0.0.0.0 = IpAddress: 10.0.2.2
RFC1213-MIB::ipRouteNextHop.10.0.2.0 = IpAddress: 0.0.0.0
RFC1213-MIB::ipRouteNextHop.192.168.200.0 = IpAddress: 0.0.0.0
RFC1213-MIB::ipRouteType.0.0.0.0 = INTEGER: indirect(4)
RFC1213-MIB::ipRouteType.10.0.2.0 = INTEGER: direct(3)
RFC1213-MIB::ipRouteType.192.168.200.0 = INTEGER: direct(3)
RFC1213-MIB::ipRouteProto.0.0.0.0 = INTEGER: local(2)
RFC1213-MIB::ipRouteProto.10.0.2.0 = INTEGER: local(2)
RFC1213-MIB::ipRouteProto.192.168.200.0 = INTEGER: local(2)
RFC1213-MIB::ipRouteMask.0.0.0.0 = IpAddress: 0.0.0.0
RFC1213-MIB::ipRouteMask.10.0.2.0 = IpAddress: 255.255.255.0
RFC1213-MIB::ipRouteMask.192.168.200.0 = IpAddress: 255.255.255.0
```

Listing 2: Obtaining a routing table using SNMPv2

Another, although not very reliable way of determining the type of a remote device is so called "banner grabbing". A service, for instance an SSH daemon, that is active and accepting connections may identify itself with a specific message or banner that it sends when a connection to it is being established. Some services often include platform information, thus making it easy to identify the target host. However, this is not always the case, banners can be modified or may contain only a standard message that identifies the service, rather than the platform it is running on. The snippets in Listing 3 represent a couple of scenarios of such messages and the information that can be obtained using this approach.

```
vits@y550 ~ $ telnet 192.168.200.10 22
Trying 192.168.200.10...
Connected to 192.168.200.10.
Escape character is '^]'.
SSH-2.0-OpenSSH_5.9p1 Debian-5ubuntu1

vits@y550 ~ $ telnet 10.0.0.1 80
Trying 10.0.0.1...
Connected to 10.0.0.1.
Escape character is '^]'.
...output omitted...
Date: Wed, 30 Jan 2013 08:21:15 GMT
Server: lighttpd/1.4.29

vits@y550 ~ $ telnet 172.16.10.10 22
Trying 172.16.10.10...
Connected to 172.16.10.10.
Escape character is '^]'.
SSH-2.0-Cisco-1.25
```

Listing 3: Service "banner grabbing" examples

The second case in the above example does not provide a lot of information about the platform, however this response can still be useful, since a given service might only be supported by a known set of platforms, thus narrowing down the range of possible types of hosts to one in this set. The first and third snippets, however, show an SSH server running on an Ubuntu server and a Cisco router respectively, and as can be seen the platform or vendor information may also be included.

Banner grabbing can be utilized in passive discovery; however, it may be only useful in shared media networks, such as for instance wireless local area networks (WLANs). For example, Listing 4 shows the actual service banner on the wire for the last snippet in Listing 3.

```
10:45:36.402660 IP (tos 0xc0, ttl 255, id 21167, offset 0, flags [none], proto
    TCP (6), length 59)
    172.16.10.10.ssh > y550.local.57274: Flags [P.], cksum 0x009e (correct), seq
        1:20, ack 1, win 4128, length 19
        0x0000:  45c0 003b 52af 0000 ff06 fc21 ac10 0a0a  E..;R......!....
        0x0010:  ac10 0a01 0016 dfba a774 95c2 c7a0 bced  .........t......
        0x0020:  5018 1020 009e 0000 5353 482d 322e 302d  P.......SSH-2.0-
        0x0030:  4369 7363 6f2d 312e 3235 0a              Cisco-1.25.
```

Listing 4: A packet with a service banner on the wire

A similar approach, but again not very reliable, is to analyze an index page from a device running a web server. This may be useful, since many configurable devices today offer a management interface over Hypertext Transfer Protocol (HTTP). Consider, for instance, Cisco or Juniper products, where the index page often contains information about the vendor and sometimes the platform that is being used. However, web servers are pretty common in networks today, hence this

approach may produce irrelevant results, and subsequently will require more in-depth analysis of these results.

The following paragraphs will give a brief introduction to different TCP/IP header fields and options that can be useful in OS discovery, specifically when using an OS fingerprinting method. The format of IP and TCP headers is shown in Figure 2 for reference.

The first interesting field is the Time to Live field of the IP packet. This field determines the maximum amount of time the packet can exist in the network. As the packet traverses the network, each node that processes the packet decreases this field by one. Once the field reaches zero the packet must be discarded. [26] The maximum possible value is 255, however, the initial value for the field is not standardized, and it is a function of the actual implementation of the TCP/IP stack. Different implementations define different TTL values. Consider the example in Listing 5.

```
vits@y550 ~ $ ping 10.0.0.201
PING 10.0.0.201 (10.0.0.201) 56(84) bytes of data.
64 bytes from 10.0.0.201: icmp_req=1 ttl=63 time=0.905 ms

vits@y550 ~ $ ping 172.16.10.10
PING 172.16.10.10 (172.16.10.10) 56(84) bytes of data.
64 bytes from 172.16.10.10: icmp_req=1 ttl=255 time=3.03 ms
```

Listing 5: TTL values example

The recommended value for the TTL is 64 [28], hence the Linux kernel uses this recommended value. However, Cisco or Sun use a value of 255. The above snippets (Listing 5) show ICMP echo replies from a Linux machine and a Cisco router respectively. Although, it is not possible to correctly identify the exact platform of the remote system simply based upon analyzing TTL values, this information may significantly narrow down the possibilities, considering that the number of hops to the target device can be estimated with at least some precision (as can be done in most cases).

Another interesting field is the TCP Window size, which defines the amount of data the target device can accept from the sender. A given OS can utilize different window sizes, making this test not very effective, unless the different values an OS can set are collected.

The TCP header sequence number field can also be analyzed. Specifically, the initial sequence number and how the target device increments this number during the communication session.

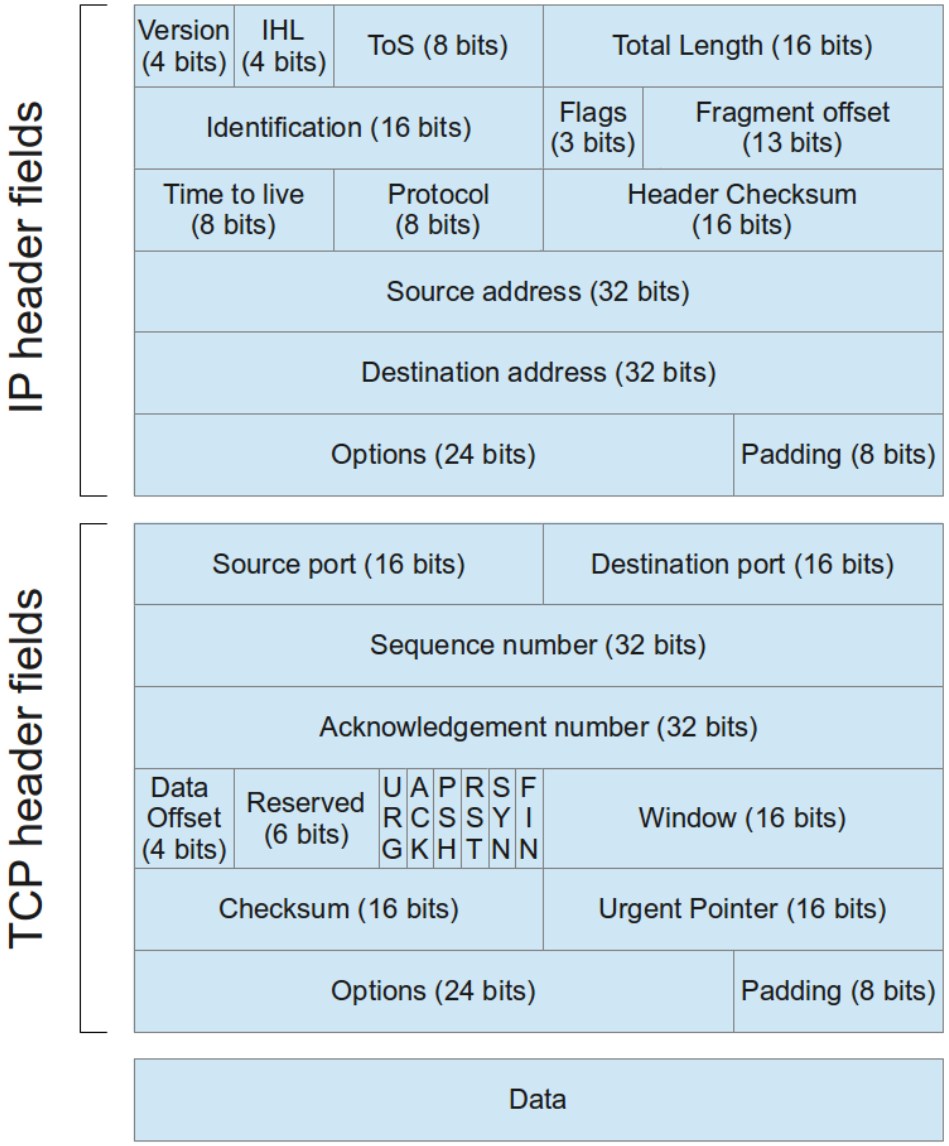

Figure 2: IP and TCP headers format [26, 27]

A number of TCP options can also be analyzed to provide additional input to the process of device discovery. One interesting option is Explicit Congestion Notification (ECN) [29] support in the TCP stack implementation. ECN-enabled hosts can signal the existence of congestion *before* starting to drop packets, so that the overall communication performance is improved. The timestamp option

and window can be scaled to improve performance by minimizing the number of retransmissions and to exploit reliable communication over high speed links [30]. These two options can also provide input to OS fingerprinting. Allen [31] describes these and several other methods in an extensive article on remote fingerprinting, which provides additional information on this topic.

Nmap uses these and many other options to create an OS fingerprint. An extensive guide to the TCP/IP fingerprinting methods supported by Nmap can be found in the official Nmap project guide [32]. Nmap maintains a large database of known fingerprints, namely **nmap-os-db**, and matches the fingerprints of discovered devices against this database. If the database does not contain an exact match of a fingerprint, an estimated match is provided along with a corresponding accuracy rating.

2.4 Link-layer discovery techniques

A special case occurs when a device is in the same broadcast domain as the management station, hence there are some additional methods that the management station can utilize to discover the device. One of the reasons is that the management station is able to exploit link-layer protocols to discover the device. Another reason is that both devices are able to receive broadcast information sent by each other.

As the management station is able to receive link-layer frames from the device, it becomes possible to discover the link-layer address of the device, for example the MAC address is readily seen on Ethernet networks. An interesting approach that may provide information for device type detection is analyzing the vendor portion or the Organizationally Unique Identifier (OUI) of the MAC address [33]. The database of OUI to vendor mapping is provided by Institute of Electrical and Electronics Engineers (IEEE) and is (mostly) publicly accessible [34]. Although a MAC address is easy to forge, generally there is no need to do so for network devices and the MAC addresses are usually used, therefore this method can be usually considered fairly reliable. Note that this will not be true for cryptographically generated addresses (CGAs) as used by IPv6 in conjunction with the SEcure Network Discovery (SEND) protocol [35], if the MAC address is to be extracted from the IPv6 address.

A simple and quick method to search for devices in a broadcast domain is to employ ARP which is intended for IP address to MAC address translation. An ARP request is issued for each IP address in each subnet which is connected to the

management station and the responses are collected. The MAC address contained in the response may be used to determine the device vendor as described above. Since the device must respond to an ARP request (otherwise it would be virtually offline), ARP network scanning provides very reliable results. For example, arp-scan [36] is a tool which performs network scanning with ARP and may be used to produce these results.

As the management station is able to receive broadcast frames sent by the device, it also makes sense to passively listen to network traffic, since the management station already receives all of these broadcast frames. Since some network devices utilize link-layer discovery protocols, these frames can be sniffed by the management station and may provide an important source of information for detecting the type of the device.

2.5 Link-layer neighbor discovery protocols

Link-layer neighbor discovery protocols are protocols which act on the link layer of the network and allow direct neighbors to discover each other. In order for devices to see each other the link between them does not have to be active, but it has to be enabled, i.e. this link must participate in routing or switching, so that the devices connected with redundant links will discover each other. These protocols not only allow discovery of the presence of a device, they also offer some information about the device, such as an upper-layer address (i.e. IP address), device model, or information about what software this device is running.

LLDP is a standardized non-vendor-specific protocol which provides neighbor discovery functionality. It is defined for the IEEE 802 protocol stack (specifically, it carries IEEE 802.1 and 802.3 related information [37]) and operates on top of the underlying MAC layer. Essentially, each participating device sends LLDP Data units (DUs) out all of its LLDP-enabled interfaces and listens for incoming LLDP DUs. This is a one-way protocol, so there is no way to request information from a particular device, even if the device is known to be a neighbor [38]. Additionally, LLDP allows a restricted mode of operation, in which the device only transmits LLDP DUs or only receives DUs.

An LLDP DU [38] consists of a MAC header specific destination MAC address, an EtherType, and a body. The body is a set of Type-Length-Value (TLV) units which contain information about the device. There are 4 mandatory TLVs and an arbitrary number of optional TLVs. The mandatory TLVs are:

- 1st TLV: Chassis ID - device hardware identifier, i.e. MAC address

- 2nd TLV: Port ID - outgoing port identifier, i.e. port number
- 3rd TLV: TTL - period of time (in seconds) during which the provided information is valid
- Last TLV: End of LLDP DU

The predecessor of LLDP was the Cisco Discovery Protocol (CDP). CDP is a Cisco proprietary protocol for link-layer neighbor discovery. It is similar to LLDP, however they are not interoperable. While both LLDP and CDP are supported by many Cisco devices, other devices support only CDP. This has to be taken into consideration when discovering the network's topology. There are a number of other proprietary link-layer neighbor discovery protocols from other vendors, such as: Extreme Discovery Protocol (EDP), Nortel Discovery Protocol (NDP), and Foundry Discovery Protocol (FDP).

Link-layer neighbor discovery protocols are essential for the task of topology discovery since it is the easiest way to detect redundant links. Other ways of doing so include studying the device configurations and the output from protocols such as the Spanning Tree Protocol (STP) or routing protocols, however utilizing this information is a much more complex task.

2.6 IPv6 discovery techniques

Some of the discovery techniques, that can be successfully utilized for IPv4, may not apply to IPv6. This is due to some significant differences between IPv6 and IPv4, such as a much larger address space and the local link address concept [39, 40]. However, IPv6 has some mechanisms that may be particularly useful for device discovery and topology discovery. The rest of this section will provide information about particular IPv6 features that can be useful in device and topology discovery.

There is no broadcast address support in IPv6. The broadcast concept was completely replaced with multicast; and since multicast is now an integral part of the protocol (while in IPv4, multicast was an extension [41]), IPv6 provides better support and structuring for multicast addressing. For instance FF02::1 and FF02::2 are the multicast addresses that identify all nodes within the link-local (or interface-local) scope and all routers within site-local, link-local or interface-local scopes respectively [42]. These multicast addresses may be utilized to discover the hosts and routers within the defined scope (interface-local, link-local, and site-local). An example in Listing 6 shows the effect of sending ICMPv6 requests to

multicast addresses accessible via the eth0 interface and the information that can be obtained.

```
victorts@nms:~$ ping6 -I eth0 ff02::1
PING ff02::1(ff02::1) from fe80::8044:61ff:fe6b:14a0 eth0: 56 data bytes
64 bytes from fe80::8044:61ff:fe6b:14a0: icmp_seq=1 ttl=64 time=0.031 ms
64 bytes from fe80::1ec1:deff:fe6d:6852: icmp_seq=1 ttl=64 time=0.383 ms (DUP!)
64 bytes from fe80::3e4a:92ff:fedb:38aa: icmp_seq=1 ttl=64 time=1.14 ms (DUP!)
64 bytes from fe80::9c45:6eff:fe26:be91: icmp_seq=1 ttl=64 time=1.23 ms (DUP!)

victorts@nms:~$ ping6 -I eth0 ff02::2
PING ff02::2(ff02::2) from fe80::8044:61ff:fe6b:14a0 eth0: 56 data bytes
64 bytes from fe80::40d5:ccff:fe5a:631a: icmp_seq=1 ttl=64 time=0.424 ms
64 bytes from fe80::215:17ff:fe76:9d1: icmp_seq=1 ttl=64 time=1.73 ms (DUP!)
```

Listing 6: IPv6 multicast example

The first snippet in Listing 6 shows all the hosts that can be reached by sending only one ICMPv6 echo request, effectively avoiding the need to "ping" each address in the subnet. The second part of the listing shows all the routers that can be found on the subnet, again by sending only one ICMPv6 packet.

The Neighbor Discovery protocol specified for IPv6 is a protocol that allows IPv6 enabled nodes that reside on the same link to discover each other, discover routers, and maintain reachability information [43]. The router and neighbor discovery mechanism relies on the multicast addresses described above. This ND protocol utilizes a set of ICMPv6 messages to enable the nodes to perform discovery related tasks. These messages include:

- router advertisement messages sent by the routers to announce their presence;
- router solicitation messages sent to request a router advertisement;
- neighbor solicitation messages sent to discover the nodes within the same scope; and
- neighbor advertisement messages as responses to the neighbor solicitation messages.

IPv6 also provides a security extension to its Neighbor Discovery protocol called SEND [44]. SEND introduces security related options to the original version of the neighbor discovery protocol. These options are intended to protect the messages. Specifically, in order to increase the level of authenticity, CGAs are used to verify the sender of a particular neighbor discovery message. While CGAs may not seem directly relevant to device discovery techniques, they may be particularly useful when detecting illegitimate router advertisement messages.

Router advertisement messages may also be particularly useful in passive discovery as they can be used to identify the routers in a given subnet.

It has to be noted that IPv6 stateless address autoconfiguration [45], is responsible for generating link-local and global addresses as well as duplicate address detection. This process relies on the neighbor discovery protocol together with its solicitation and advertisement messages.

The next chapter discusses one of the thesis project's tasks, namely network device discovery. This discussion includes a description of the task, the proposed solution, and its implementation.

Chapter 3

Device discovery implementation

The first task for this master's thesis project was to develop a module for NCS which performs network device discovery. The module should be able to identify which devices in the specified address space are alive and be able to provide some information about the device, such as device vendor, device model, and OS running on the device. This chapter will describe the requirements defined for the device discovery module and the process of developing this module.

3.1 Device discovery module description

The device discovery module for NCS is a component for automated discovery of network devices within a given IP address range. This module should provide the following functionality: discover the devices that are online and automatically load a selected device from the set of discovered devices into the NCS operational database with the most complete set of parameters that can be determined during the discovery process. The module must integrate with the NCS data model and run within the NCS Java Virtual Machine (VM). The module has to be developed in the form of an NCS package which can be optionally loaded into NCS at startup.

An important part of the requirements for the module was to focus on the devices that can be managed by NCS, as opposed to finding *all* devices that are online. Also, as the module is a legitimate search tool, it was assumed that network security systems are properly configured to allow the operation of this tool, therefore the solution proposed should **not** contain any techniques of firewall, Intrusion Detection System (IDS)/Intrusion Prevention System (IPS), or other

network security systems bypass. Since the discovery process is assumed to be run by the network administrator or in coordination with the network administrator, it is also assumed that the user of the network device discovery module knows the credentials to access the legitimate devices.

The device discovery module consists of two parts: a data model described in YANG and executable code written in Java. The data model is a set of instructions for NCS which defines the format of input and output parameters for the operations performed by the module, as well as the format of the data stored by the module in the NCS database. In turn, NCS renders its user interface (CLI and WebUI) to the module based on this data model. The executable code receives input parameters in the specified format and performs the necessary actions to obtain a set of output parameters. These output parameters are then passed to the NCS in the specified format. The code can also store necessary data in the NCS database in the specified format. This data can later be used by the module itself or accessed by an NCS user.

Among the initial requirements from Tail-f was to use Nmap to provide device discovery functionality for this module. The reason behind this was that Nmap is a widely used network scanning tool and it implements a wide variety of known device discovery techniques. Additionally, relying on Nmap for device discovery was expected to significantly reduce the development time for this module. However, during an early phase of development, it appeared that additional techniques are difficult to integrate with an Nmap-based solution; furthermore, the company decided that a monolithic package would be more convenient to ship to the customers than a package which includes an Nmap distribution. So, eventually it was decided to focus on a stand-alone implementation of the device discovery functionality.

The first version of the package was Nmap-based, but was complemented by a stand-alone device discovery engine. The final version of the discovery module contains both solutions and allows a user to choose between the two options. Although the device discovery functionality is implemented differently, both solutions share the same data model. The next section discusses the data model of the package.

3.2 Data model description

The data model of NCS is defined in YANG, which makes it extensible by incorporating additional modules into the main model. The data model of the

network device discovery component augments the NCS data model and adds a module that defines a set of actions with associated input and output parameters as well as a structure for storing the data collected when a specific action is executed. An action, in this context, is an interface to call and execute specific code defined for that action. The interface is rendered by the NCS CLI and WebUI, so that the action becomes an executable command that can be invoked via the user interface (CLI or WebUI). An action may have input and output parameters that are also modeled in YANG. Thus, the parameters may be well structured and the format of the data predefined. The device discovery component's data model defines five actions: two actions to launch the device discovery process (one for the Nmap-based component and one for the stand-alone discovery component) and three actions to process the list of discovered devices: an action to select a specific device and load this device into the NCS configuration database, an action to remove a specific device from the list of discovered device, and an action to load all discovered devices at once into the NCS configuration database. These actions provide the required flexibility for an operator when loading discovered devices into the configuration database.

The input parameter for the discovery launch actions is a definition of the discovery target, which may be a single IP address (e.g. **10.0.0.24**) or a domain name (e.g. **example.com**), an IP subnet specification (e.g. **10.0.1.0/28**), an IP range specification (e.g. **10.0.2.14-29,82**), or a set of targets in any of these forms (e.g. **[172.16.2.0/24 192.168.25.4-29 10.0.0.2]**). This set of target specification options provides the required level of flexibility to specify the target IPv4 address space. The output parameter for these actions is a status line which indicates if the run was successful or if there were any errors. An IPv6 target specification is supported in the Nmap-based discovery module; but support for IPv6 in the stand-alone module is left as future work.

The actions used to either select and load or remove a specific device from the set of discovered devices, or select and load the whole set of discovered devices are `"pick"`, `"forget"`, and `"pick-all"` actions. The `Pick` action is associated with the list of discovered devices and is called from the context of the selected device allowing an operator to set additional input parameters. Thus, an operator can optionally provide a new name for the device, an authentication group, etc. when loading the device into the CDB. The `Forget` action is also called from the context of the selected device which is to be removed, this action does not have any input parameters. The `Pick-all` action is called from the **discovery** subtree and allows the operator to set some common parameters for all the loaded devices. As designers we believe that the `pick-all` action is flexible when used in combination with the `forget` action (i.e., an operator can explicitly remove those devices that are not relevant and then quickly add all the remaining devices). The

output parameter for each of the actions is a status line which is used to inform the user as to whether the operation was successful or some error occurred.

Since the main function of the device discovery module is to discover network devices and present the discovered devices to the NCS operator, the data model should incorporate additional structures which will define a model to store these results. Thus, additional structures defined in the model include a list of devices found and their associated parameters, such as device vendor, device description, open ports, and other information that could be obtained. The list of devices is defined as operational (non-configurable) data (marked with **config false** in YANG terms), this means that this data can not be manipulated via the user interface (CLI or WebUI) and it has a better visual representation than configurable data. Defining the list of devices as operational data allows us to use a single API both to fetch and store the values for the parameters defined in the model.

Additionally, the model contains a data structure to define pattern matching rules for device discovery and a container for storing various credentials (currently SSH usernames and associated passwords, SNMPv1 and SNMPv2 community strings, and SNMPv3 credentials). The pattern matching data structure and credentials container are only utilized in the stand-alone discovery module. Configurable pattern matching rules provide a flexible way of defining platform and service detection rules and representation with additional rules being provided directly by NCS. Adding a list of credentials allows the stand-alone module to actually connect to a discovered device and, thus obtain the exact information about the device's type, which can not be done in the Nmap-based implementation (although Nmap provides estimated matches for the OS detection, an exact match is not always possible). Since the stand-alone module is treated as a tool for the NCS operator, rather than a network scanner, it is assumed that some set of credentials are known to the operator, and, thus better results can be obtained than with Nmap. The pattern matching data structure and credential lists are defined as configuration data, which can be set by the operator using one of the user interfaces.

The stand-alone component utilizes some of the techniques mentioned in Chapter 2, thus it can provide useful results even if credential lists are not defined. In addition to the configurable set of credentials, the model contains a container for the valid credentials (the "working" credentials from the set of configured credentials, i.e., those that the NCS is currently using for its operations) that are associated with a specific device; thus, the operator can see which of the credentials were valid for that device. This set of valid credentials is configured as operational data.

The flexibility of YANG, among its other capabilities, is that it supports derived data types. YANG allows a developer to create custom data types that rely on the standard data types but may incorporate restrictions, such as pattern matching based restrictions, length of string values, range of numerical values, etc. This allows us to validate the parameters directly at input time, so that the code does not have to implement validation methods, as the parameter values were already validated within the model itself. Notable derived data types defined for the device discovery model are IP range specifications and SNMP OID string specifications; both utilize pattern matching.

The device discovery data model defines the structure for all the data that is associated with the device discovery component, and, thus, serves the foundation of the component. The specifics of the stand-alone and Nmap-based modules are described in the following sections of this chapter.

3.3 Implementation based on Nmap

The first version of the network device discovery module was essentially a wrapper for Nmap. This module operates as follows: receive the input parameters, execute Nmap with predefined flags and the specified target, wait for Nmap to finish, parse the results of the Nmap scan, and store the results in the NCS database according to the model. The output generated by the module contains informational messages, such as the number of devices found and the success or failure of the execution. The execution is considered successful when there are no errors while running Nmap and no errors while processing the results or loading them into the database, even if no devices are found in the specified address space.

The flags and parameters for Nmap execution were chosen according to the module's functional requirements. The major protocols for network device management defined for the device discovery component are NETCONF, SSH, Telnet, HTTP, and SNMP. So it was decided to limit the set of protocols to be checked for to these protocols. To reduce the complexity of the scan and to reduce the scan time it was assumed that these protocols run on their default ports. Therefore the list of ports to be scanned was: 22/tcp, 23/tcp, 80/tcp, 443/tcp, 830/tcp, and 161/udp. The corresponding Nmap parameter is "**-p T:22-23,80,443,830,U:161**".

Other flags instruct Nmap to try to obtain the desired information about the device. With the "**-O**" and "**–osscan-guess**" flags Nmap will perform OS detection and try

to guess the OS the device is running (note that an exact match is not required, but database entries with the closest fingerprint pattern are displayed). With the "**-sSU**" flag Nmap will perform TCP SYN port scanning, as well as User Datagram Protocol (UDP) port scanning. It is worth mentioning that among all the different types of TCP port scanning, we are only interested in a TCP SYN scan since we are looking for the ports that NCS should later be able to connect to. The "**-sV**" flag instructs Nmap to try to determine the service version for every open port, which is essentially a banner grabbing technique. This information can also be useful for determining the OS running on a device. The "**-T4**" flag tells Nmap to perform an aggressive, but not insane scan. In Nmap the terms aggressive and insane scans are assumed to be the two fastest types of scans. The aggressive scan is preferred due to its lower overall scan time, without sacrificing detection accuracy. The insane scan may cause inaccurate OS detection (according to the Official Nmap Project Guide [32]). Finally, "**–system-dns**" flag instructs Nmap to use the Domain Name System (DNS) server configured in the system.

Initially, Nmap tries to detect open and closed TCP ports on the scanned device by sending a TCP connection request to each of the ports. Information about the port state is used in subsequent tests. To perform OS detection (invoked with the "**-O**" flag) Nmap uses the TCP/IP stack fingerprinting technique, which was described in Section 2.3. However, when service version detection is enabled ("**-sV**") the information obtained from the ports contributes to OS detection. During the service version detection phase Nmap establishes connections to all the open TCP ports and performs banner grabbing, either without probes or using very simple probes. An example of a probe used by Nmap to perform HTTP service version detection, as captured by tcpdump, is presented in Listing 7. UDP ports should be scanned using probes according to the protocol which is assumed to be running on the corresponding port, e.g. a set of SNMP requests is sent to UDP port 161 in order to determine if the port is actually open and to obtain information about the SNMP version.

```
12:20:48.615336 IP denis-laptop.tail-f.com.47983 > moon.tail-f.com.http: Flags [P
    .], seq 1:19, ack 1, win 115, options [nop,nop,TS val 2137094 ecr 489454175],
     length 18
        0x0000:  0018 51f6 24f4 60eb 69af f9fc 0800 4500  ..Q.$.`.i.....E.
        0x0010:  0046 33f0 4000 4006 82b6 c0a8 018f c0a8  .F3.@.@.........
        0x0020:  012c bb6f 0050 9e94 a5c7 ef0c 2840 8018  .,.o.P......(@..
        0x0030:  0073 8444 0000 0101 080a 0020 9c06 1d2c  .s.D...........,
        0x0040:  7a5f 4745 5420 2f20 4854 5450 2f31 2e30  z_GET./.HTTP/1.0
        0x0050:  0d0a 0d0a
```

Listing 7: HTTP service probe used by Nmap

Nmap provides different output options, including normal output, XML output, and "grepable" output. XML output was chosen as it is easy to parse

automatically, and specifically the ease of extracting the required values from the output based on their tag names. The "**-oX**" flag is used to specify XML output. The output flag requires a filename of the output file as the parameter. This filename is generated as a random string of 26 letters in the Java code. The complete set of flags used for Nmap is shown in Listing 8 together with the output filename (following the "**-oX**" flag) and the address range (10.0.0.0/24).

```
/usr/bin/nmap -PN --system-dns -O --osscan-guess -sSUV -T4 -p T:22-23,80,443,830,
    U:161 -oX /tmp/4jc7sh7ago1m0cur5tih8pfct5.nmaprun 10.0.0.0/24
```

Listing 8: A set of Nmap flags used for the device discovery component

After Nmap finishes the scan, the device discovery module processes the XML file produced by Nmap and fetches the scan results. These results are then stored in memory for further processing. For this purpose a `Device` class was defined which consists of a set of properties describing a network device found during the device discovery process. Additionally, this class contains methods for storing and loading the information about the device into the NCS database according to the defined data model. When the list of discovered devices is fetched, the **latest-run** subtree of the data model is replaced with the new list of discovered devices. The information is stored as operational data, which means that it will be reset when NCS is restarted. This information is displayed to the user and can be further processed by another application.

The implementation based on Nmap was a good starting point as it allowed rapid development of a working solution. However, Nmap does not have the abilities to use a predefined set of credentials to access the devices and the precision of device type detection in Nmap based solution is not always satisfactory (to meet the requirements of NCS). From this perspective a stand-alone engine for network device discovery offers greater flexibility and more implementation options. The following section discusses the stand-alone discovery module's design and implementation.

3.4 Stand-alone device discovery engine

The stand-alone device discovery engine is implemented in Java as a separate action within the network device discovery module. It provides discovery of network devices and provides information about the device (make, model, and OS the device is running) in a way that is more efficient, more precise, and better satisfies the requirements of NCS than an Nmap-based solution.

3.4.1 General logic of the stand-alone device discovery engine

The general logic of this engine is the following. First, the input parameter, which is a target specification, is processed by the `Parser` object, which forms a list of `Device` objects with only the IP address or addresses selected. Then a `Scanner` object is created for each of the devices in a separate thread. This `Scanner` object performs the scanning process, i.e. applies the scanning techniques to the defined IP address(es). If a device is discovered to be offline, it is removed from the list of devices, otherwise the `Scanner` populates the data fields in a `Device` object with all the information that it is able to discover about the device. Lastly, the list of discovered devices is stored in the NCS database as operational data for further processing.

As an input the `Parser` object takes a target specification in the form described in Section 3.2. First, it checks if the target matches the input IP address prefix regular expression. If the target does not match, then the `Parser` checks if it matches an IP address range regular expression, which also allows a single IP address. If the target matches either an IP prefix pattern or an IP range pattern, then the `Parser` iterates through all of the possible values for this prefix or this range, creates an "empty" `Device` object (which has only an IP address specified) and forms a list of `Devices` including all of the `Device` objects that have been created. If the target does not match any of these options, then the `Parser` assumes that the target is a hostname and tries to resolve its IP address. If resolution succeeds, a one-entry list which contains an "empty" `Device` is created, otherwise the `Parser` fails and the module halts with an "Invalid target specification" error. In case of success, the `Parser` returns the list of `Devices` as an output.

3.4.2 TCP port scanning

The `Scanner` object takes a `Device` as an input. The `Scanner` has a set of predefined TCP ports to scan. Some of the TCP ports are assumed to run specific services, for example the `Scanner` expects SSH to be on port 22 and HTTP on port 80. Initially the `Scanner` performs a port scan iterating through the set of expected TCP ports. To scan a TCP port the `Scanner` tries to initiate a connection using a socket whose remote endpoint is defined by the `Device`'s IP address and the port to be scanned. Figure 3 illustrates all the possible outcomes of this connection, which are essentially a successful connection or an exception. If there is an exception, then the port is considered closed; however, the device state may result in: "Up", "Possibly down", or "Down". If the port is in the open or closed

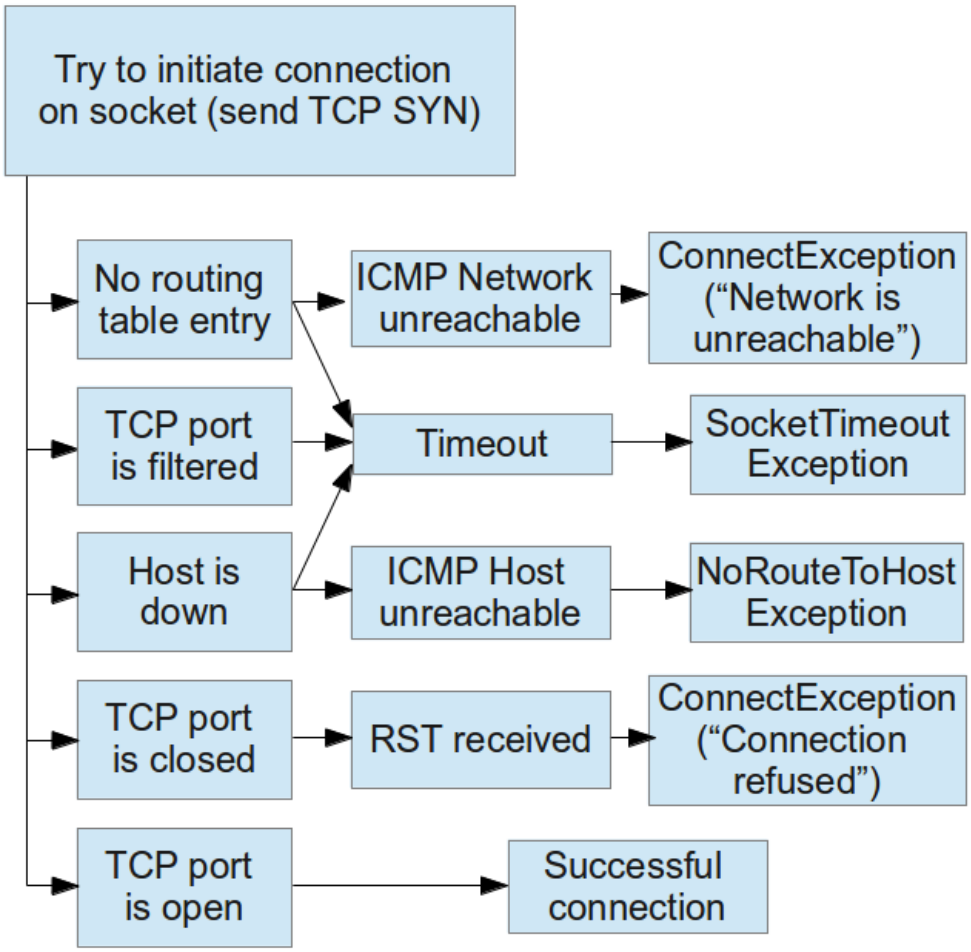

Figure 3: Possible outcomes of TCP port scanning attempt

state, the device is considered to be "Up"; if any expected ICMP error message is received, then the device is considered to be "Down" and a timeout will not change the state of the device, i.e. a port remains in the "Possibly down" state if it is not known to be "Up" yet.

If the connection is successful, then the `Scanner` will try to communicate with the device using the protocol which is assumed to be running on this port. A set of classes which implement a `Communicator` interface are used for this purpose. Currently there are communicators to communicate with a device via SSH, HTTP, and Hypertext Transfer Protocol Secure (HTTPS) protocols. There is a `DefaultCommunicator` which performs simple banner grabbing: it does not send any probes, but listens for a service to send data which identifies the service.

If the service is not expected to send any data and there is no communicator for the service, then the `VoidCommunicator` is applied which immediately closes the port.

The `HttpCommunicator` object is created when a connection is successfully established on one of the ports which are assumed to run HTTP. This object implements a very basic HTTP client which sends simple HTTP/1.1 GET requests and follows HTTP redirects. If a response is received, then the `HttpCommunicator` applies a set of regular expressions to it. Specifically, we look at the response code and **Location** header to follow a redirect if there is any; we also try to fetch some information about the device by looking at the **Server** header, **WWW-Authentication** realm, and the contents within the **title** tag.

The `HttpsCommunicator` object takes as an input a socket which is connected to an HTTPS port. It attempts to establish an Secure Sockets Layer (SSL)/TLS connection with the device and in case of success an `HttpCommunicator` is created which communicates with the device via the newly created SSL socket.

The `Scanner` employs an `SSHCommunicator` to communicate with devices which have an SSH port open. Many SSH implementations send a banner in clear text when the connection is established. This banner gives some information about the SSH implementation and sometimes contains some information about the OS running on the device. However, as we assume that an NCS operator has the credentials to access the device, we will try to connect to the SSH port. The SSH client was not implemented from scratch, but rather the **Ganymed SSH-2** library was used. Credentials to access the device are retrieved from the NCS database. The current version of the module supports only password-based authentication. There is a possibility to specify several username and password pairs, if different devices in the network need to be accessed using different credentials. We assume that the NCS is not blocked by a network security system even if there are several unsuccessful connection attempts.

Once `SSHCommunicator` succeeds in being authenticated, different commands are issued to obtain information about the device vendor, model, and OS version. An output of a specific command usually matches some pattern of the devices made by the same vendor, so that `SSHCommunicator` is able to extract the required information from the command's output with regular expressions, e.g. the output of the **show version** command for Cisco devices is described in Cisco's IOS Command Reference [46] and the output of the same command for Juniper devices is described in the Juniper technical documentation [47]. For devices running Linux the **uname** command may be used. The output format of this command is described in the Linux manual [48]. If no command matches, then

`SSHCommunicator` reports that the connection was successful and specifies the credentials that are found to be valid for this device, but does not give any extra information about the device.

If in any point during the `Scanner` operation the device appears to be in the "Down" state, then the `Scanner` immediately halts and reports that the device is offline. However, if the device appears to be in the "Up" state, then the scanning proceeds until all of the TCP ports are scanned, because the goal of the TCP scanning process is to obtain as much information about the device as possible, including the list of management services that the device is running. If the device is in the "Possibly down" state after scanning all the TCP ports, i.e. all the connection attempts resulted in a timeout, then the `Scanner` attempts to send an ICMP Echo Request to identify whether the device is up. Strictly speaking, not receiving an ICMP Echo Response does not mean that the device is offline; however, if the device did not respond to any of the TCP connections and does not respond to an ICMP Echo Request, then most probably this device can not be managed with NCS. That is why if the device does not respond, then it is considered to be in the "Down" state. If the device responded to at least one of the TCP connections or sent an ICMP Echo Response, it is considered in the "Up" state and the `Scanner` proceeds with UDP port scanning.

3.4.3 SNMP scanning

In contrast to TCP port scanning, UDP port scanning is different due to the connectionless nature of the UDP protocol. However, in the context of our NCS discovery module we are most interested in only one UDP based protocol, namely SNMP (as NCS can manage SNMP enabled devices when certain conditions are met). For the purposes of the device discovery module, we are particularly interested in obtaining the system description string, which is available in MIB-II and is usually present in most devices that are SNMP enabled.

Generally, a UDP port will respond (or not respond; this is protocol specific) only to probes that match the actual protocol listening on that port, making TCP port scanning techniques (such as banner grabbing) inapplicable. Thus, in order to check an SNMP port, SNMP specific probes are required. However, an important aspect of sending the SNMP probes is to provide the correct credentials, i.e. the community strings for SNMP version 1 and version 2, and login credentials for SNMP version 3. For this purpose a credentials structure is defined in the data model (see Section 3.2).

For the stand-alone device discovery engine, it was decided to perform SNMP port scanning *after* a device was successfully determined to be online, as it is quite unlikely that a device will have all other ports (in the set of scanned ports) filtered, and not respond to an ICMP echo request. This allows us to decrease the overall scanning time. Once the `Scanner` finishes a TCP port scan and reports a device to be alive, the SNMP port scanning is initiated. The SNMP related functionality is implemented in the `SnmpDevice` sub-class which extends the `Device` class.

When the SNMP scan is initiated a new `SnmpDevice` object is created from the `Device` object that is currently being processed. The `Scanner` iterates through the configured set of community strings for SNMP version 1 and version 2, and sends GetNextRequest Protocol Data Units (PDUs) [49, 50] for the system description object (1.3.6.1.2.1.1.1) defined in MIB-II. If the system description is successfully obtained, then a valid community string (the one that was successfully used to obtain the result) is stored in the valid-credentials data structure and the information about the port is stored in the `Device` object. If scanning via SNMP version 1 and 2 fails, then the `Scanner` tries scanning via SNMP version 3. The scanning logic is the same: the `Scanner` iterates through the set of configured SNMPv3 credentials and sends a request for a system description object.

The port along with associated information is added to the `Device` object if the scanning is successful and the valid set of credentials is stored. On devices with SNMPv3 support it is actually possible to determine if the port is open, since SNMPv3 will respond with a specific MIB object when invalid credentials or parameters (authentication and privacy algorithms) are used [51]. In this case, if there is no valid credential set among the preconfigured credentials then the port is reported as open with additional information that the credentials were not valid. The scanning is performed sequentially: first over version 1 and 2; and second over version 3. In some cases a device may support two versions of the protocol (version 1 or 2 and version 3); if this is the case, system description object is fetched via both versions, and the device is reported to support both versions of SNMP.

SNMP scanning functionality is implemented using the SNMP4J [52] API that ships within the standard NCS package. The SNMP4J library handles the low level details of SNMP communication, in particular it manages timeouts and connection retries, as well as processes certain type of exceptions, such as an ICMP Port Unreachable message [53].

3.4.4 Storing device type and description

After the scanning is completed and a list of `Device` objects (which contains devices that were discovered along with their associated information which the `Scanner` was able to retrieve) is created, then the type, description, and vendor of the device are determined. This information is extracted from the data that was discovered during port scanning. First, a vendor name or a set of keywords associated with the vendor (such as Juniper Operating System (JunOS)) is searched for in the port information field of the `Device` object. It is assumed that one vendor would not use the name of another vendor in the services their device provides. Next, each port information field in a `Device` object is processed and matched to a regular expression set that matches expected `SSHCommunicator` module output as well as some of the well-known SNMP system description formats; in this way the hardware platform and the OS information are obtained and stored in the respective fields of the object. Once these fields are set, the device type is determined. Determining the device type and setting appropriate management port and management protocol are essential requirements for a device to be successfully stored in the NCS CDB, as well as to enable the NCS to be able to subsequently manage that device.

Currently, there are four device types in the context of NCS management interfaces:

- NETCONF devices that are NETCONF enabled and can be managed over this protocol;
- CLI devices that implement a Cisco style command line interface;
- SNMP devices that do not support the above protocols, but can be managed via SNMP; and
- generic devices that require customized interfaces.

CLI and generic devices require specifically designed NEDs loaded into NCS and SNMP devices require MIB modules represented in YANG, while NETCONF devices are natively supported by NCS. The type of a device is determined based on the information that was obtained during scanning. The NETCONF device type is set if the device is detected to be a Juniper device (as Juniper devices support NETCONF by default [54]) or if TCP port 830 (NETCONF over SSH [8]) is open on the device. The CLI device type is set if the device is reported to be a Cisco device (at the moment of writing this thesis, Cisco devices are the only devices that can be managed by NCS over a CLI management interface).

The management ports to use are determined using a port priority algorithm which matches the open ports discovered on a device to a set of ports defined in the algorithm which are listed in order of preference. The TCP port priority order for NETCONF devices is 830, 22, and for CLI devices: 22, 23. The first matched port that is open is set as the management port for this device.

When a device is **not** detected as a NETCONF or CLI device, then the SNMP port is checked. If UDP port 161 is reported as open, then the device type is set to SNMP. All other devices discovered not to support any of the above mentioned types are identified as generic devices.

Once the list of discovered devices is processed and the type and management protocol are set, the devices are stored in the operational database. The next section will describe the process of selecting a specific device from the set of discovered devices and loading this device to the NCS CDB.

3.5 Loading devices into NCS

The `pick` action implements the functionality of fetching the full set of information related to a discovered device from the operational data and storing the device into the NCS configuration database. Once a device is stored in the CDB as a managed device, NCS can connect to the device and manage it. The `pick` action is defined within the device list structure of the discovery component data model. Thus, the action is bound to a specific device in the list of discovered devices. Calling the action from the context of a device allows us to offer an NCS operator the option to provide additional parameters when they select a device, such as assigning a new name or authentication group.

Since NCS requires specific NEDs for CLI and generic devices, the `pick` action provides an option to select a specific NED according to the device type. However, NCS can not manage a NED device without the respective NED, thus, if there are no NEDs matching the device's type, then this device can not be stored in the CDB. In this case an operator has to load a corresponding NED prior to loading the device into the NCS configuration.

To allow an operator to load multiple devices with a single command, the `pick-all` action has been implemented. It acts essentially in the same way as a `pick` action, but is called from the main **discovery** container (as opposed to the `pick` action which is called from the device context) and loads all of the devices in the discovery list into the configuration database. This action only allows setting the same NED and the same authentication parameters for all the loaded devices.

Additionally, it allows the operator to customize automatically detected device names by prepending a specified name prefix. The `forget` action enables the operator to remove a single device from the discovery list, which offers a lot of flexibility in combination with the `pick-all` action.

As a part of its functionality, the `pick` action scans for supported MIBs on a device, when the device is manageable via SNMP and the credentials for accessing it are known. The mapping of MIB names to the matching OIDs is stored in the NCS database and defined in the data model. The software sends a GetNextRequest PDU with the OID for each of the MIBs listed in the database. After receiving an answer from the device, the software matches the OID in the response with the requested OID. If the object in the response is located within the requested subtree, it is assumed that the device supports the requested MIB. In the current version of the module, the list of supported MIBs is displayed to the NCS operator, but it is planned to use this information for setting SNMP management parameters for the device when adding it to the NCS CDB.

3.6 Device discovery module architectural overview

The device discovery module consists of two main components: a data model defined in YANG and program logic implemented in Java. The package is a loadable module which augments and adds additional data structures into the NCS data model. The logical overview of package integration is presented in Figure 4. The data model is defined as a module with included submodules which makes the model more flexible and easy to follow. The program logic is a set of Java classes, which consists of base and extended classes reflecting the required package functionality.

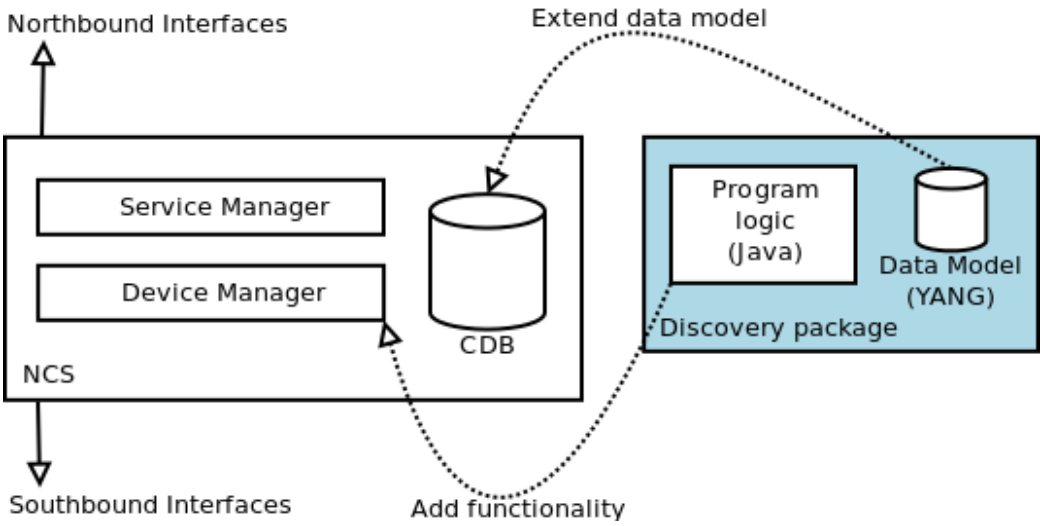

Figure 4: Discovery package logical overview

The data model for the discovery package is a YANG module which consists of several submodules. The submodules help to organize the data structures according to their functionality in order to make the module easier to extend and organize. The feature of including a submodule within another submodule in the context of the same parent module allows reuse and referring to data structures within submodules. Figure 5 represents the structural overview of the data model defined for the discovery package. The complete structure of discovery data model is presented in Appendix A.

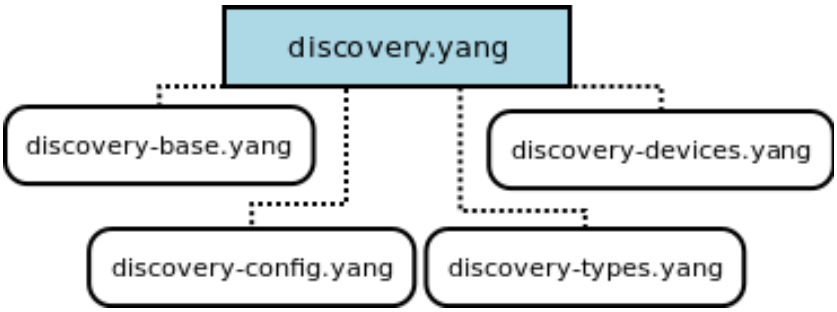

Figure 5: Discovery data model overview

The architectural overview of the Java classes for Nmap-based discovery, stand-alone discovery, and loading devices into NCS is presented in Figures 6, 7, and 8 respectively. The complete architectural view of the Java classes is presented in Appendix B.

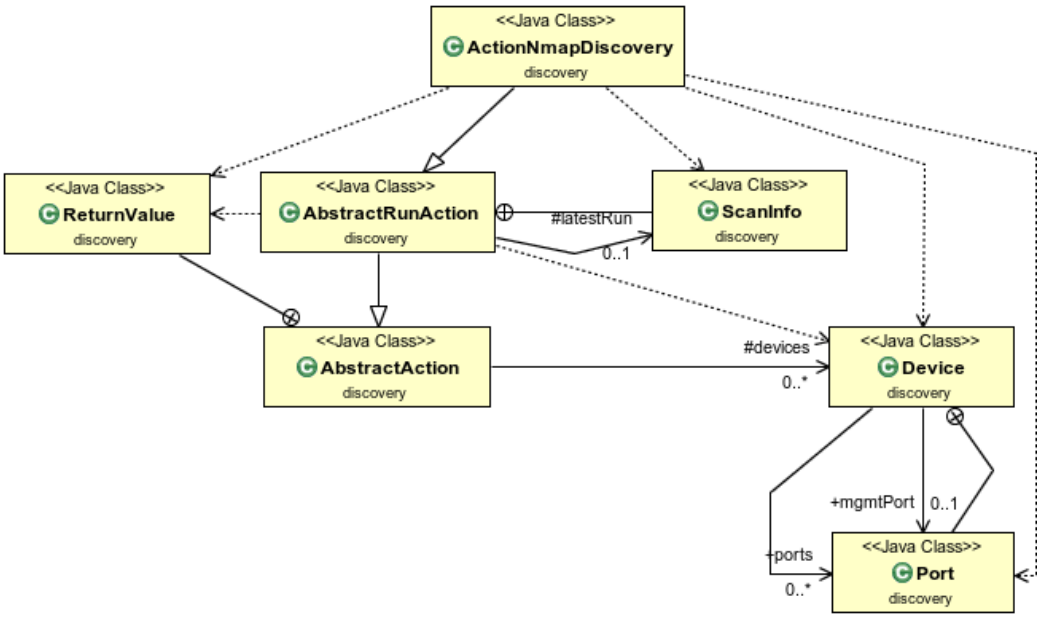

Figure 6: Nmap-based discovery architectural overview

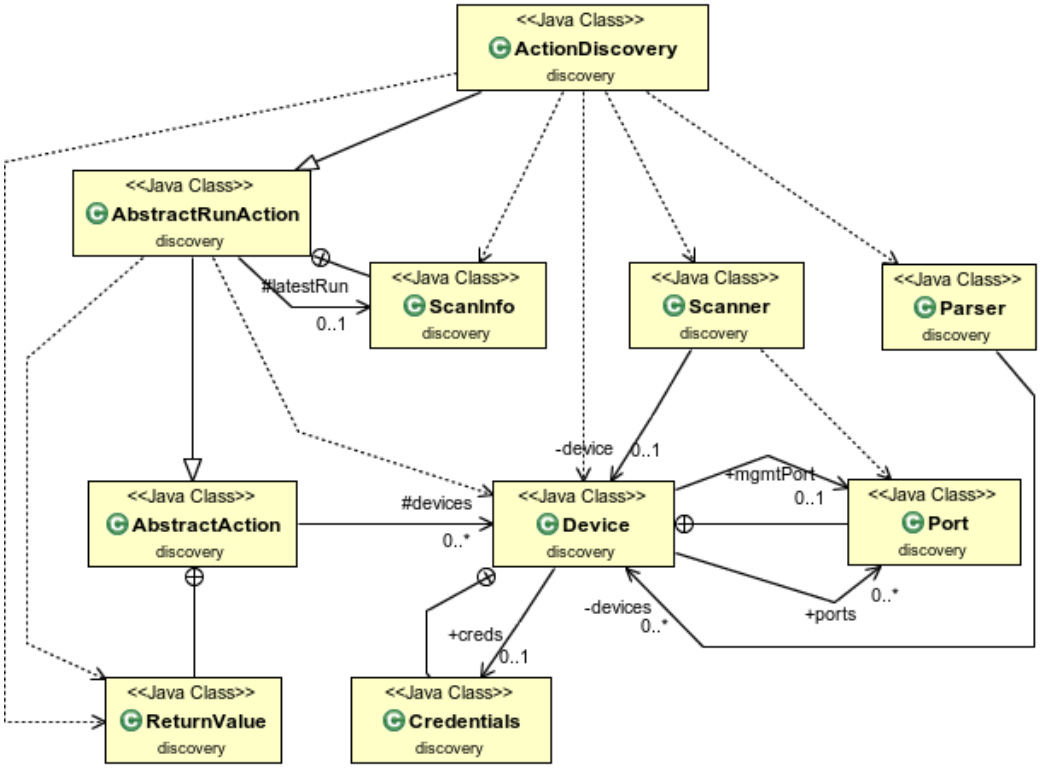

Figure 7: Stand-alone discovery architectural overview

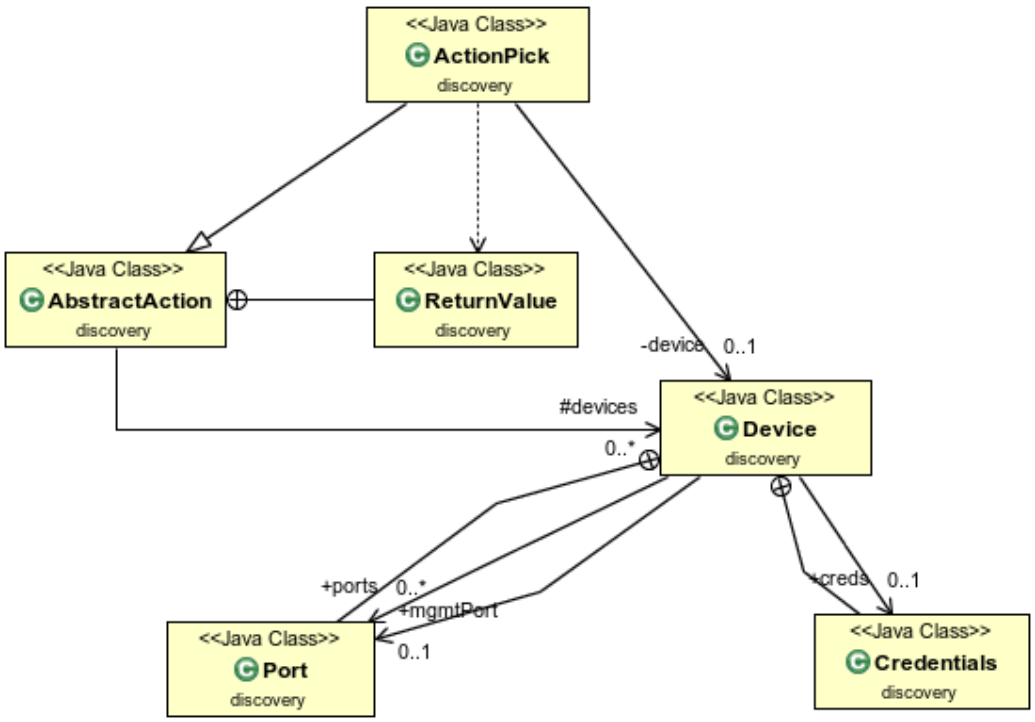

Figure 8: Loading devices into NCS architectural overview

3.7 Results and analysis

The outcome of the device discovery component design and implementation is an optional package for NCS. This package provides automated network discovery functionality and the possibility to add newly discovered devices to the NCS CDB. The package's functionality minimizes the effort of the network operator when adding a device by automatically loading the set of parameters that were obtained during the discovery process (device type, management port, management protocol, etc.). The device discovery module implementation utilizes two different approaches:

- Nmap-based device discovery module
- A stand-alone device discovery module

The rest of this section will give examples of usage of this module and associated results. A comparison between Nmap-based and stand-alone discovery functionality and performance will be presented, as well as an analysis of important aspects of the design and its implementation.

In keeping with the initial requirements from Tail-f, the first version of the discovery module was solely Nmap-based. Relying on Nmap to provide device discovery functionality allowed us to focus on learning about the NCS internals and the APIs which it provides; this knowledge made the subsequent stand-alone module design and implementation more straightforward. For the Nmap-based module the primary goals were to choose an appropriate set of flags to use for the Nmap scan, to become familiar with the internal structure of NCS and its packaging concept, and most importantly understand the CDB fetching and storing operations. The Nmap-based module is a part of the device discovery package and can be used as soon as the package is loaded to the NCS system. Listing 9 shows a typical invocation of the Nmap-based module with some input parameters and output which summarizes the result of the discovery process (as it is executed from the NCS CLI).

```
admin@y550> request devices discovery nmap-scan list-target [10.0.0.21,201
    172.16.0.2]
result
Nmap successfully finished
XML successfully parsed. Number of devices fetched: 3
Devices successfully stored as operational data
ok
[ok][2013-03-08 15:00:45]
```

Listing 9: Nmap-based discovery input and output example

Once the scan is completed, various **show** commands may be used to view the information about the devices discovered and some scan statistics. Listing 10 shows the statistics of the scan presented in Listing 9, and Listing 11 provides an example of the devices which were discovered. The scan's "duration" shown in Listing 10 is in units of seconds.

```
admin@y550> show devices discovery latest-run-info
devices discovery latest-run-info params "/usr/bin/nmap -PN --system-dns -O --
    osscan-guess -sSUV -T4 -p T:22-23,80,443,830,U:161 -oX /tmp/5
    v5jf5669bpbbj1unrq6iujdet.nmaprun 10.0.0.21,201 172.16.0.2"
devices discovery latest-run-info time "Fri Mar  8 15:00:45 2013"
devices discovery latest-run-info hosts-up 3
devices discovery latest-run-info hosts-down 0
devices discovery latest-run-info duration 30.08
[ok][2013-03-08 15:18:29]
```

Listing 10: Statistics of an Nmap-based discovery run

```
admin@y550> show devices discovery device | notab
devices discovery device 172.16.0.2
 address            172.16.0.2
 management-port    22
 management-proto   ssh
 device-description "Generic. Probably:Cisco 2821, 6506, or 7206VXR router (IOS
     12.2)"
 device-type        cli
 port 22
  port-info "ssh Cisco SSH 1.25 "
 port 23
  port-info "telnet Cisco router "
 port 161
  port-info "snmp SNMPv1 server "
devices discovery device kepler.lab
 address            10.0.0.21
 management-port    161
 device-description "Generic. Probably:Linux 2.6.15 - 2.6.26"
 device-type        snmp
 port 22
  port-info "ssh OpenSSH 5.1p1 Debian 5 "
 port 80
  port-info "http Yaws httpd 1.77 "
 port 161
  port-info "snmp SNMPv1 server "
devices discovery device 10.0.0.201
 address            10.0.0.201
 management-port    830
 device-description "Juniper Networks JUNOS 9.0R2.10"
 device-type        netconf
 port 22
  port-info "ssh OpenSSH 4.4 "
 port 161
  port-info "snmp SNMPv3 server "
 port 830
  port-info "ssh OpenSSH 4.4 "
[ok][2013-03-08 15:29:45]
```

Listing 11: Example of discovered devices in Nmap-based discovery

The Nmap-based implementation uses the following techniques to identify the type of the scanned device:

- port scanning;
- service probing and banner grabbing; and
- OS fingerprinting.

The stand-alone discovery module has the same user interface for invoking the discovery process and for storing statistical information. The results, however, differ significantly as additional data model structures (mainly the credential sets) are utilized. Listing 12 presents an example of the stand-alone module's discovery as performed with the same input parameters as were used in Listing 9; it also shows the statistics of this action.

```
admin@y550> request devices discovery scan list-target [ 10.0.0.21,201 172.16.0.2
     ]
result
Input parameters parsed: 3 devices to be scanned
Devices scanned. Alive: 3
Devices successfully stored as operational data
ok
[ok][2013-03-08 16:01:29]
admin@y550> show devices discovery latest-run-info
devices discovery latest-run-info params "fast run : target = 10.0.0.21,201
    172.16.0.2"
devices discovery latest-run-info time "Fri Mar 08 16:01:18 CET 2013"
devices discovery latest-run-info hosts-up 3
devices discovery latest-run-info hosts-down 0
devices discovery latest-run-info duration "10.75 sec"
[ok][2013-03-08 16:01:37]
```

Listing 12: Stand-alone discovery input and statistics

As it can be seen from Listing 10 and Listing 12 stand-alone discovery module is faster (10.75 seconds) than Nmap-based implementation (30.08 seconds). There are several reasons for this. Nmap utilizes internal timing algorithms which do not take advantage of the additional data sets embedded in our implementation. The only timing option we utilize for Nmap scans is the "**-T4**" option, which does not specify the fastest and most aggressive scan in Nmap. Moreover, Nmap has internal algorithms that adapt the number of probes sent based upon network conditions (network bandwidth, security measures, etc.). The stand-alone module is very straightforward from this perspective; it simply sends probes to devices from parallel threads (with the default number of threads set to 20). However, since the stand-alone module exploits credential based discovery it produces different results. The discovery results for the same input parameters as in Listing 9 are presented in Listing 13. Listing 13 also shows the valid credentials, that actually worked and were used to retrieve the information (the actual usernames, passwords, and community strings have been edited out for this example).

```
admin@y550> show devices discovery device | notab
devices discovery device 172.16.0.2
 address            172.16.0.2
 management-port    22
 management-proto   ssh
 device-vendor      Cisco
 device-description "HW:7200 IOS:12.4(7h)"
 device-type        cli
 port 22
  port-info "MSG:SSH-2.0-Cisco-1.25 ; INFO: Cisco IOS Software, 7200 Software (
      C7200-JK9O3S-M), Version 12.4(7h), RELEASE SOFTWARE (fc1)"
 port 23
  port-info "telnet port is open"
 port 161
  port-info "SNMPv1 or SNMPv2; INFO: Cisco IOS Software, 7200 Software (C7200-
      JK9O3S-M), Version 12.4(7h), RELEASE SOFTWARE (fc1)\r\nTechnical Support:
      http://www.cisco.com/techsupport\r\nCopyright (c) 1986-2007 by Cisco
      Systems, Inc.\r\nCompiled Thu 18-Oct-07 23:33 by stshen"
devices discovery device kepler.lab
 address            10.0.0.21
 management-port    161
 device-vendor      "Debian"
 device-description "OS:Linux"
 device-type        snmp
 port 22
  port-info "MSG:SSH-2.0-OpenSSH_5.1p1 Debian-5"
 port 80
  port-info "200; SRV:Yaws/1.77 Yet Another Web Server; TITLE:Index of /"
 port 161
  port-info "SNMPv1 or SNMPv2; INFO: Linux kepler 2.6.32-bpo.5-openvz-amd64 #1
      SMP Fri Jun 11 09:56:18 UTC 2010 i686"
devices discovery device 10.0.0.201
 address            10.0.0.201
 management-port    830
 device-vendor      Juniper
 device-description "HW:olive JUNOS:10.3R2.11"
 device-type        netconf
 port 22
  port-info "MSG:SSH-1.99-OpenSSH_4.4 ; INFO: HW:olive ; Juniper ; JUNOS:[10.3R2
      .11]"
 port 161
  port-info "SNMPv3; INFO: Wrong credentials"
 port 830
  port-info SSH-1.99-OpenSSH_4.4
[ok][2013-03-08 16:28:22]
admin@y550> show devices discovery valid-credentials | notab
devices discovery valid-credentials device 172.16.0.2
 ssh username *username*
 ssh password *password*
 snmp-community *community*
devices discovery valid-credentials device kepler.lab
 snmp-community *community*
devices discovery valid-credentials device 10.0.0.201
 ssh username *username*
 ssh password *password*
[ok][2013-03-08 16:29:45]
```

Listing 13: Example of discovered devices in stand-alone discovery

As it can be noted, the stand-alone module is very effective in the context of an NCS discovery component, if the correct credentials are configured. However, if the credentials are not configured or not valid, then the Nmap-based implementation may provide better results in terms of detection (but these devices will require valid credentials before NCS would be able to manage them). The purpose behind developing the stand-alone discovery module was to offer functionality that does not rely on third-party components and can be shipped as a monolithic package, and most importantly to integrate additional functionality related to using credentials for discovery purposes. Thus, the appropriate device discovery techniques were chosen from those described in Chapter 2 and successfully applied together with exploiting credentials that can be used for discovery, and ultimately producing more specific results with regard to device type detection.

In comparison with the Nmap-based implementation, the stand-alone module does not use the OS fingerprinting technique, as it would require a deep study of the different TCP/IP stack implementations used in networking equipment. Additionally, this technique provides little help in differentiating between different devices from the same vendor, which occurs when testing the Nmap-based module with different devices from the same vendor. A possible reason behind this is that the same TCP/IP stack implementation is often used in different devices from a single vendor.

On the other hand, the stand-alone implementation uses a more advanced service probing technique for HTTP/HTTPS ports (the module processes not only the Server header, but also the WWW-Authentication realm - which often provides the name of the device and the contents of the title tag) as well as SSH and SNMP ports, since the module can utilize its knowledge of the credentials to access the device it can fetch an exact device description and learn the exact version of the OS the device is running.

Once devices are discovered, it is possible to add them to the NCS CDB. As a result NCS can connect and actually manage the devices. The `pick` action described in the previous section is presented in Listing 14. This listing also shows how the selected device looks when it is added to the NCS CDB.

```
admin@y550> request devices discovery device 172.16.0.2 pick name router1
Value for 'cli-ned-id' [c7200]: c7200
result
Device loaded from operational database
Device router1 stored in the running database
ok
[ok][2013-04-11 12:11:53]
admin@y550> show configuration devices device router1
address   172.16.0.2;
port      22;
authgroup default;
device-type {
    cli {
        ned-id   c7200;
        protocol ssh;
    }
}
source {
    added-by-user admin;
    context       "cli (added by discovery package)";
    when          2013-04-11T10:11:53.193+00:00;
    from-ip       127.0.0.1;
}
[ok][2013-04-11 12:12:16]
```

Listing 14: Adding a device to the NCS CDB

As mentioned earlier, the stand-alone module is usually more effective both in terms of performance and precision (when used with valid credentials). The performance of the stand-alone engine is improved due to the use of different discovery techniques. The stand-alone engine does not implement fingerprinting techniques, thus the time to scan each device is reduced by 6 round-trip times (Nmap sends a series of six TCP probes for its TCP/IP fingerprinting implementation [32]). However, credential-based discovery creates additional overhead, especially when the configuration contains a large number of incorrect (or inapplicable) credentials.

The scalability of the both implementations is similar. When used with large numbers of devices (significantly larger than the number of threads) the overall execution time grows linearly with the average time spent per single device. The average time required to scan a device depends on several factors: whether the number of offline devices in the specified range is large, whether the devices' ports are filtered, the number of incorrect credentials in the list, the implementation of the device's TCP/IP stack (which affects OS fingerprinting time in the case of the Nmap based module). Due to variability of these factors it is impossible to make a direct numerical comparison of the execution time of these two implementations.

The main advantage of the stand-alone engine over the Nmap-based engine is its ability to perform credential based discovery. Although Nmap can guess

the device's hardware or the version of the running OS with high probability, in most cases it can not provide an exact match. Making use of the known credentials enables the stand-alone engine to be able to guarantee the correctness of the fetched result. However, since stand-alone engine is **not** supposed to act as a general-purpose network scanner, the accuracy of identification the device's parameters without knowledge of the credentials is lower than the accuracy of the Nmap-based module.

A significant disadvantage of the Nmap-based module is the need for an extended license for Nmap usage, as the default Nmap license [1] does not allow using any of the Nmap source code nor executing Nmap and parsing the results in a non-General Public License (GPL) product. The stand-alone discovery engine eliminates this need for extended license, thus reducing the cost of the end product for the company, while providing more relevant functionality than the Nmap-based engine.

Among other disadvantages of Nmap-based module is its need to be run with superuser privileges, primarily needed in order to be able to use raw sockets to perform OS fingerprinting. In contrast, the stand-alone engine can be run by an unprivileged user as most of its functionality is implemented with conventional sockets. The only action that requires privileged access is checking for the host's availability with ICMP Echo Request/Echo Response, but this action does not have a critical impact on the results produced and can be omitted. A possibility to run device discovery as an unprivileged user can be useful, as it enables an NCS operator with restricted access to the host system to be able to run the discovery process if this action is allowed for him or her by the NCS's security policy.

The following chapter will discuss the design and implementation of a topology discovery module. It describes the approach used to develop the module and results of this implementation.

Chapter 4

Topology discovery implementation

The second task in this master's thesis project was to develop an NCS module to discover the topology of the network which consists of the devices managed by NCS. The module should be able to identify both the logical network map and physical connections between the network devices. As previously stated, it is assumed that the topology discovery is performed by a network administrator or on his or her behalf, so the network devices can be controlled by an NCS operator. This chapter discusses the requirements and the development process of the topology discovery module.

4.1 Topology discovery module description

The goal of the topology discovery module is to identify both logical and physical topologies of the network. The topology discovery module should receive a set of devices as an input. For the logical topology, the module should be able to identify routing capabilities of the devices (which of the devices act as routers) and to identify all the subnets (network addresses and network masks) connected to the routers. For the physical topology, the module should be able to identify physical links between the network devices wherever possible. The logical topology discovery process should be able to identify connections between routers or Layer 3 (L3) switches (that is why such a logical topology is also referred to as an L3 topology). For parts of the logical topology (usually, a single subnet identified by L3 topology discovery) the topology discovery module should provide a detailed view with the help of physical topology discovery, whose primary goal is to show

links between Layer 2 (L2) switches (hence the physical topology is also referred to as an L2 topology). The module must integrate with the NCS data model and run within the NCS Java VM. The module has to be developed in the form of an NCS package which can be optionally loaded into NCS at startup.

The initial idea was to develop a module which would perform topology discovery based on results of the device discovery module described in the Chapter 3. After creating an initial prototype and discussing its operation with the company, it was decided to perform topology discovery based on the devices configured in NCS. On one hand, this decision extends the flexibility of the input set of devices as not only discovered devices can be used, but also existing devices that are currently managed by the NCS. On the other hand, it leverages NCS capabilities for retrieving the necessary data from the device which, in turn, eliminates the need to manage the list of credentials which are necessary to access the devices and it also avoids the need to maintain a communication channel with each of the devices. Additionally, it removes a dependency on the device discovery module.

A part of this task is visualization by creating a network map, i.e. a graphical representation of the topology discovery results. This representation should allow for easy integration with the NCS WebUI. However, the integration with the WebUI is outside the scope of this project, thus the visualization is only described and implemented as a prototype at this stage.

The topology discovery package has its own YANG data model and executable Java code. The description of both the model and the code is presented in the following sections.

4.2 Data model description

The data model of the topology discovery module is defined in YANG. It consists of 3 submodules: **topology-base**, **topology-l2**, and **topology-l3**. The first submodule defines the position of the top node of the topology discovery subtree in the main NCS data model tree. The latter two submodules define data models for the physical and logical topology discovery results respectively.

Each of the submodules defines one action and a data structure to store the results of the operation. Both actions receive one input parameter which identifies the scope of operation. Both actions have an option to scan over all of the devices configured in NCS or over a group of devices defined in NCS. Additionally, the physical topology discovery action has an option to specify the scope of the discovery in the form of a specific subnet discovered by the logical topology

discovery. This satisfies the requirement for providing a detailed view of parts of the L3 topology. Both actions output a status line and save the result of their operation as operational data in the CDB.

The data structure for storing the logical topology consists of a list of discovered nodes and a list of discovered subnets. Each of the nodes in the list of nodes contains the following information:

- the name of the node
- the list of this node's IP address(es)
- the list of the subnets connected to this node
- the routing capabilities of this node (router, host, or undefined)

Each of the subnets in the list of subnets is bound to the list of nodes which have an IP address in this subnet. This information is given explicitly for convenience.

The data structure for storing the physical topology consists of one list of nodes. Each of the nodes in this list contains the following information:

- the name of the node
- the identifier (system name) of this node
- the list of neighbors connected by a direct link to this node; this list includes both the local and remote interfaces used to connect these devices

The data model of the topology discovery module defines an interface to the topology discovery actions and the necessary data structures. This interface is implemented in Java. The next section describes the implementation of the L3 topology discovery action.

4.3 Logical topology discovery

The logical topology discovery action is defined in the `ActionL3Topology` class. The general logic of this action is as follows. First, the scope of the discovery is identified by parsing the parameter of the action. Then the set of devices that corresponds to the scope parameter is loaded from the NCS CDB. After this, a set of data needed for building the topology is retrieved from each of the devices using native NCS means for communication with the devices. The retrieved data is stored in data structures in memory which correspond to the

data structures defined in the data model (see Section 4.2). Finally, this data is stored in the NCS CDB. Additionally, the data is also stored in a temporary file used by a visualization library for generating a graphical representation of the topology.

The necessary data is retrieved from each device via SNMP. However, this communication with the device is performed by means of NCS, so the topology discovery module does not have to implement any SNMP capabilities, instead it simply reads a corresponding part of the data tree for each particular device. The retrieved data is not cached in CDB, so each time that NCS receives a request for a particular piece of operational data, it fetches the data directly from the device using SNMP.

To compute the topology we use the following data from the devices:

- IP addresses of the interfaces (both physical and logical).
- Network masks corresponding to the IP addresses.
- Operational state of the interfaces corresponding to the addresses. Inactive interfaces do not participate in building the network map.
- IP forwarding variable, which allows the code to identify routers (we assume that every node which has IP forwarding enabled is a router).

Using this data it is possible to calculate a network address for each of the subnets directly connected to the node. Given a list of nodes (which is the input for this action) we can create a list of subnets directly connected to these nodes. As we know the connections between the nodes and the subnets, it is possible to build a network map, essentially a graph with two types of nodes: devices (routers and hosts) and subnets. The action does not implement visualization though, but only stores data in a temporary file in a format compatible with the visualization library (see Section 4.5).

It is worth mentioning that this implementation is non-vendor-specific, as it does not rely on any proprietary MIBs. Each device should support the IF-MIB [55] and the IP-MIB [56] and be configured as an SNMP device in NCS or have SNMP configured as a secondary protocol for gathering operational data from the device. If a device does not satisfy these requirements, then it may appear on a network map as a host node in the case its management IP address falls into the address range of any of the discovered subnets. No other data collection methods have been implemented (for now); the main reason for this is the lack of data models for fetching operational data via SSH (CLI) and NETCONF in the current version of NCS.

4.4 Physical topology discovery

Physical topology discovery is represented as an action defined in the L2 topology data model. The general characteristics of the physical topology discovery resemble those of the logical topology discovery; however, this action relies on different data sets. The action is implemented using the `ActionL2Topology` class. There are several input choices defined for this action: all configured devices from the NCS managed devices tree, a group of managed devices, or a subnet from a previously run L3 topology discovery action. As in the logical topology discovery, the physical topology discovery relies on the internal mechanisms of NCS to retrieve the operational data via SNMP, thus, there is no need to implement any communication specific features within the action. Once the input is specified the L2 discovery process queries specific fields within the operational data tree of a specific device, then retrieves and stores the data as the operational data according to the L2 topology model (as described in Section 4.2).

Physical topology discovery relies on the CDP protocol to produce a list of nodes with associated neighbors connected to the nodes. Thus, the current version of the physical topology discovery supports only CDP enabled devices. The `CDPCacheTable` from the Cisco-CDP-MIB [57] is queried to fetch the required information:

- The device's global identifier (ID) is used as an identifier set for the device (usually represented as the device's domain name);
- The list of neighbors connected to this device with their respective global identifiers, and
- The local and remote interface names for the device and associated neighbor.

Once the data is fetched, it is stored as operational data in the CDB. The data is stored as a list of nodes with an associated list of neighbors and the information about the interfaces. Thus, each node has a global ID and a list of connected neighbors with respective local interface (interface on the node itself) and remote interface (interface on the neighbor). This data is saved into a temporary file which will subsequently be used by the visualization library to generate a graphical representation of the physical topology.

Initially, it was planned to also utilize LLDP for building L2 topology maps; however, due to the limited availability of hardware with LLDP support, an LLDP based implementation was not realized in the context of this thesis project (but should be considered for future work). Nonetheless, possible approaches to

utilizing LLDP were discussed and some implementation logic was developed. The general approach is to follow the same logic as described in the CDP based topology discovery: requesting the LLDP related data through NCS internal mechanisms over SNMP. As the implementation may need to support devices from different vendors, we have assumed that the specific SNMP MIBs for the LLDP protocol are available from these vendors. Another approach, which is more flexible and easier to implement, but unfortunately is not supported on many devices, is to utilize the NETCONF Remote Procedure Call (RPC) mechanism. This RPC mechanism could be used to request operational data from a device. Juniper devices are a good example of such devices. The advantage of this approach is that it eliminates the need for using SNMP, since the native NETCONF functionality can be utilized. For Cisco devices a similar approach could be developed using Cisco's CLI to communicate with the device. However, since there are no native data models for collecting operational data via NETCONF and Cisco's CLI in NCS (for now), these data models must be developed to enable the topology discovery module to retrieve operational data from the devices via these protocols.

4.5 Topology visualization

Topology visualization aims to provide a graphical representation of the topology discovery data that has been collected. As NCS provides an extensible WebUI, it was decided to implement this visualization using the **Arbor.js** [58] JavaScript library to allow further integration of the visualization implementation with the WebUI. However, the actual integration of visualization with WebUI is outside the scope of this thesis project, thus, only a possible prototype of the visualization implementation was developed to offer a base for further integration efforts.

The logic of the topology visualization implementation is straightforward: once the L2 or L3 topology discovery action is completed and the discovered data is stored in the operational database, then the data is processed and temporary files to be used by **Arbor.js** are produced. These files contain descriptions of the nodes and the edges (links between the nodes) in the format used by this library. The library than renders the data from generated files and produces a visual representation. A good feature of **Arbor.js** is that the data may be stored with additional parameters, such as colors, shapes, and labels; and the rendering engine may be modified to reflect the requirements of the visualization. As a result, the actual topology data is completely separated from the visualization logic.

Since it was decided to implement the visualization in a way that will potentially facilitate further integration of the visualization into the NCS WebUI, the choice of the visualization library had to correspond to that requirement. There are different graph visualization libraries and software available, including the widely used Graphviz [59], that could have been used. However, the requirement for this library is to provide an easy solution to integrate with the WebUI, including possibility to link the nodes in the visual topology representation to the actual device representation in the NCS WebUI and scalability to different screen resolutions that would be difficult to implement with static images. Hence the JavaScript based approach has been taken that would provide all of the required features. While Graphviz is very flexible relying on the DOT language [60] which provides a good way of defining graphs, Graphviz generates static images as an output, which can not be further manipulated. There is a Graphviz based JavaScript library which also relies on the DOT language - Canviz [61]; this is a very promising library, however, there is no support for features like mouse-over and mouse-out events, animations, and draggable interface for now. There is also a range of different JavaScript graph visualization libraries, and **Arbor.js** is one of those. The choice for **Arbor.js** was motivated by the good design of the library and by allowing us to rapidly develop a prototype for the visualization, however, this is still an open issue, and the final choice for the library should be based on the NCS WebUI integration requirements.

It is worth mentioning that the temporary XML file produced by Nmap during the Nmap-based device discovery may be used to provide topology visualization of devices discovered during the Nmap device discovery process. Nmap uses RadialNet [62] to create visualization of discovered devices' topology. RadialNet is very good at producing visualizations, however, it is a tool that was developed specifically for Nmap which makes its use with other software problematic. Since the stand-alone device discovery engine has been chosen as a primary engine due to the reasons described in Section 3.7, the topology visualization approach based on Nmap was not pursued.

4.6 Topology discovery architectural overview

The topology discovery module follows the structure of the device discovery module. It consists of two components: the data model and the program logic. The module represents an NCS package which can optionally be loaded into NCS on startup. Moreover, the topology discovery package relies on other NCS packages to provide support for the required functionality by fetching the required data

using SNMP. The dependencies are verified at NCS startup, and the package is disabled if the packages on which it depends are not loaded. A logical overview of the package integration is presented in Figure 9. The packages on which this module depends provide required functionality for the topology discovery package; specifically the IP-MIB package allows retrieval of the IP addressing information, while the Cisco-CDP-MIB package allows retrieval of the CDP related information.

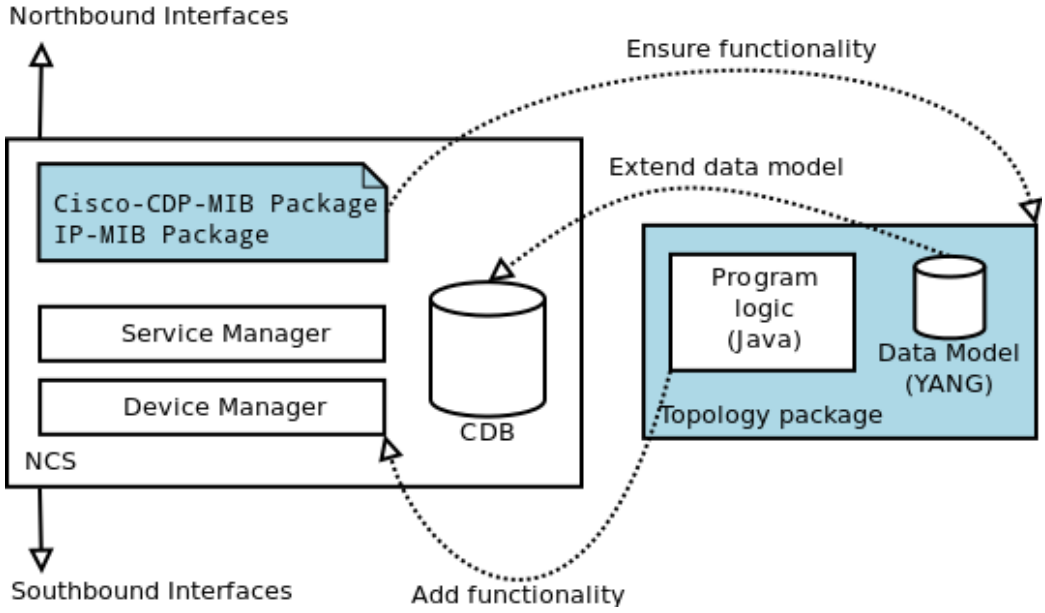

Figure 9: Topology package logical overview

The data model of the topology discovery package is defined as a YANG module with included submodules. Figure 10 represents the structural overview of the topology module. The functionality of the submodules was described in Section 4.2. The complete structure of the topology discovery data model is presented in Appendix C.

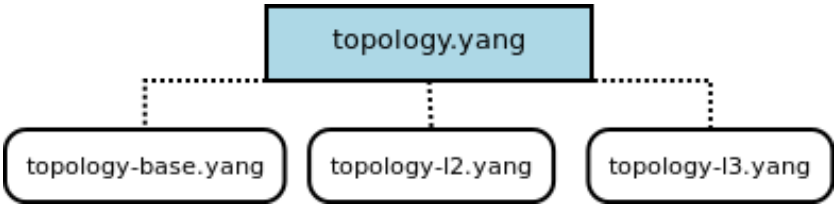

Figure 10: Topology package data model overview

An architectural overview of the Java classes for L3 topology discovery and L2 topology discovery is presented in Figures 11 and 12 respectively. The complete representation is provided in Appendix D.

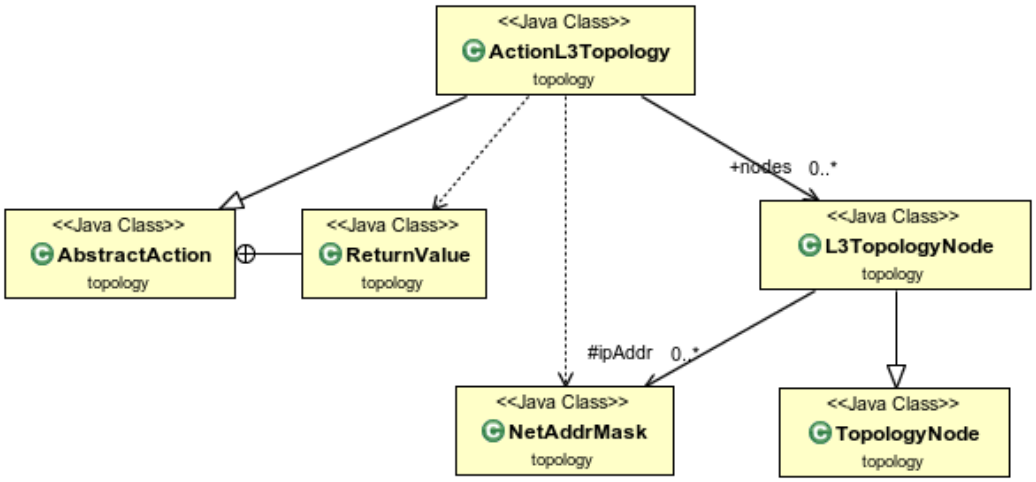

Figure 11: L3 topology discovery architectural overview

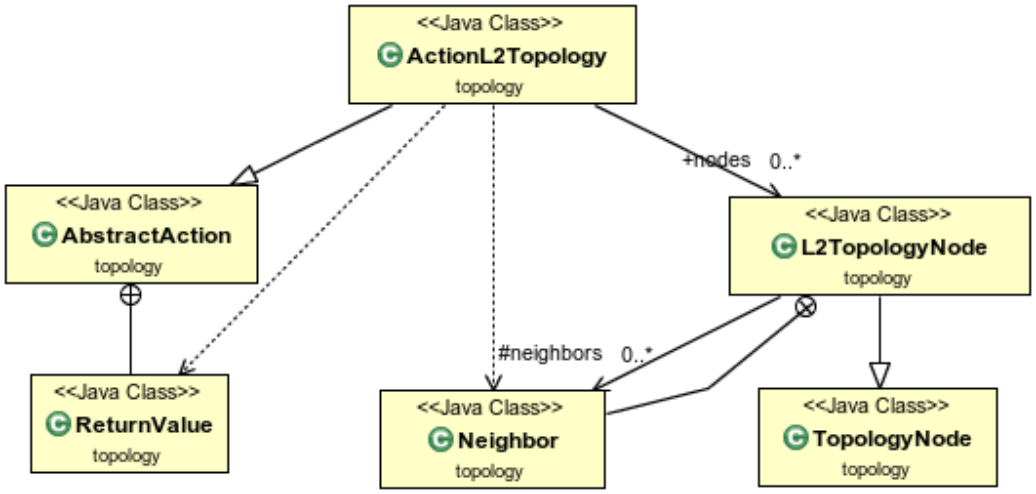

Figure 12: L2 topology discovery architectural overview

An architecture of the proposed solution was designed to be extensible in terms of the protocols which could be used for communication with the different devices. Although the current version of the module is capable of retrieving the necessary data from devices only over SNMP, the same data could be retrieved over SSH (CLI) or NETCONF. As the actual communication with the device is performed

by NCS, retrieving the data via a different protocol would be just a matter of requesting a different object from NCS. When NCS has the necessary data models for retrieving operational data via these protocols, then this functionality could be exploted by making minor changes to the source code. Similarly, it is easy to build an L2 topology using other LLDP protocols as they can provide similar functionality by collecting additional data from other MIBs and to provide additional OIDs in the source code.

4.7 Results and analysis

The topology discovery functionality was realized as an NCS package. This package provides a representation of a network topology map based on the devices in NCS configuration database. This representation includes a logical (L3) topology, which represents logical structure of the network and a physical (L2) topology representing the physical links between devices. This package also contains a prototype implementation of a module to generate a visual representation of the topology discovery process' results. The rest of this section provides a number of examples of the usage of the topology discovery module, discusses the results produced by topology discovery process, and shows examples of topology visualization, as well as mentions some important aspects of the process of developing this module.

The primary goal for the functionality of this package was to develop an appropriate approach for network topology representation as a data structure and designing an appropriate YANG data model for it. Since the logical and physical topologies of the same network may differ, it was decided to provide both representations in the same package. The logical (L3) topology is based on the actual IP address configuration of the devices, thus, the most suitable protocol for gathering this data is SNMP (as it provides the greatest coverage of different devices that we considered). However, for the physical (L2) topology, since the data about direct link connections is not readily available from the devices, additional protocols were required. As described in Section 2.4, some of the link-layer discovery protocols that could be used are CDP and LLDP, as these protocols are the most used protocols (in the context of those devices that can currently be managed by NCS). For this reason, the L2 topology discovery implementation relies on the CDP protocol for collecting the link-layer interconnection data from the devices and utilizes SNMP to collect the interconnection data from these devices by calling upon NCS. LLDP support was not included in the package at this stage due to the unavailability of equipment

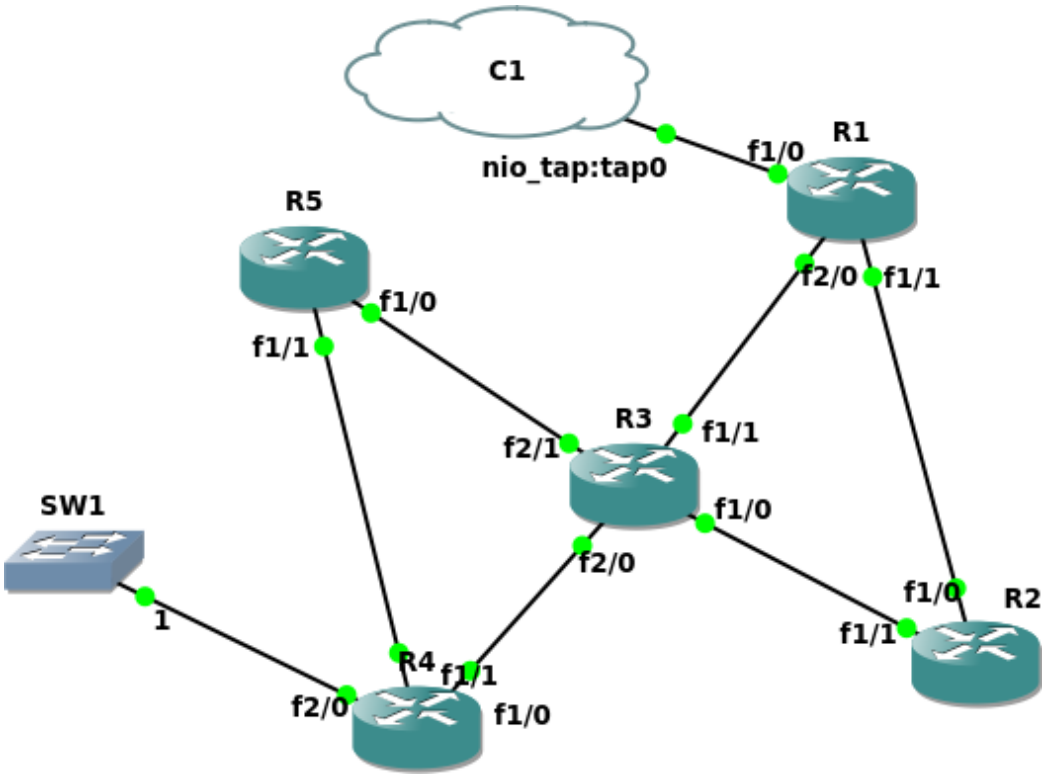

Figure 13: Virtual network topology

with LLDP support (it is planned that a later effort would target Juniper equipment for LLDP based topology discovery). Providing a visual representation of the discovered topology was a secondary goal.

The following listings and results of topology discovery are based on the virtual network presented in Figure 13. This network was specifically created for evaluating the topology discovery processes. This virtual network consists of five emulated Cisco routers. Listing 15 shows the devices currently present in the NCS managed devices tree that are used in these examples.

```
admin@y550> show configuration devices | match device | except device-type
device cis1 {
device cis2 {
device cis3 {
device cis4 {
device cis5 {
[ok][2013-05-06 11:09:52]
```

Listing 15: List of configured devices used in the topology discovery examples

L3 and L2 topology discovery actions may be run independently of each other; however, the L2 topology action includes an option to perform link-layer discovery based on a subnet that was previously discovered during an L3 topology discovery process. A typical invocation and the output of L3 and L2 topology discovery actions is presented in Listing 16 and Listing 17 (respectively).

```
admin@y550> request devices topology layer3 scan
result
Devices loaded: 5
Nodes discovered: 5
Successfully stored 5 nodes
Layer 3 topology visualization data successfully stored
ok
[ok][2013-05-06 11:11:13]
```

Listing 16: L3 topology discovery action

```
admin@y550> request devices topology layer2 scan
result
Devices loaded: 5
Nodes discovered: 5
Successfully stored 5 nodes
Topology visualization data successfully stored
ok
[ok][2013-05-06 11:12:02]
```

Listing 17: L2 topology discovery action

The results of the actions shown in Listing 16 and Listing 17 are presented in Listing 18 and Listing 19 (respectively). Both L3 and L2 topology discovery actions generate a temporary file that is subsequently used to produce a graphical representation. The temporary file is generated after the results are parsed.

```
admin@y550> show devices topology layer3 | notab
devices topology layer3 subnets 172.30.0.128/25
 nodes [ cis1 ]
devices topology layer3 subnets 172.30.10.0/30
 nodes [ cis1 cis2 ]
devices topology layer3 subnets 172.30.20.0/28
 nodes [ cis1 cis3 ]
devices topology layer3 subnets 172.30.30.0/28
 nodes [ cis2 cis3 ]
devices topology layer3 subnets 172.30.40.0/27
 nodes [ cis3 cis4 ]
devices topology layer3 subnets 172.30.50.0/29
 nodes [ cis3 cis5 ]
devices topology layer3 subnets 172.30.60.0/30
 nodes [ cis4 cis5 ]
devices topology layer3 subnets 172.30.70.0/24
 nodes [ cis4 ]
devices topology layer3 subnets 172.30.99.1/32
 nodes [ cis1 ]
devices topology layer3 subnets 172.30.99.2/32
 nodes [ cis2 ]
devices topology layer3 subnets 172.30.99.3/32
 nodes [ cis3 ]
devices topology layer3 subnets 172.30.99.4/32
 nodes [ cis4 ]
devices topology layer3 subnets 172.30.99.5/32
 nodes [ cis5 ]
devices topology layer3 nodes cis1
 addresses [ 172.30.0.130/25 172.30.10.1/30 172.30.20.1/28 172.30.99.1/32 ]
 networks  [ 172.30.0.128/25 172.30.10.0/30 172.30.20.0/28 172.30.99.1/32 ]
 type      router
devices topology layer3 nodes cis2
 addresses [ 172.30.10.2/30 172.30.30.1/28 172.30.99.2/32 ]
 networks  [ 172.30.10.0/30 172.30.30.0/28 172.30.99.2/32 ]
 type      router
devices topology layer3 nodes cis3
 addresses [ 172.30.20.2/28 172.30.30.2/28 172.30.40.1/27 172.30.50.3/29
     172.30.99.3/32 ]
 networks  [ 172.30.20.0/28 172.30.30.0/28 172.30.40.0/27 172.30.50.0/29
     172.30.99.3/32 ]
 type      router
devices topology layer3 nodes cis4
 addresses [ 172.30.40.2/27 172.30.60.1/30 172.30.70.1/24 172.30.99.4/32 ]
 networks  [ 172.30.40.0/27 172.30.60.0/30 172.30.70.0/24 172.30.99.4/32 ]
 type      router
devices topology layer3 nodes cis5
 addresses [ 172.30.50.2/29 172.30.60.2/30 172.30.99.5/32 ]
 networks  [ 172.30.50.0/29 172.30.60.0/30 172.30.99.5/32 ]
 type      router
[ok][2013-05-06 11:12:31]
```

Listing 18: L3 topology discovery results

```
admin@y550> show devices topology layer2 | notab
devices topology layer2 nodes cis1/172.30.99.1
 device-global-id R1.test.dom
 neighbor R2.test.dom 172.30.10.2 FastEthernet1/1 FastEthernet1/0
 neighbor R3.test.dom 172.30.20.2 FastEthernet2/0 FastEthernet1/1
devices topology layer2 nodes cis2/172.30.99.2
 device-global-id R2.test.dom
 neighbor R1.test.dom 172.30.10.1 FastEthernet1/0 FastEthernet1/1
 neighbor R3.test.dom 172.30.30.2 FastEthernet1/1 FastEthernet1/0
devices topology layer2 nodes cis3/172.30.99.3
 device-global-id R3.test.dom
 neighbor R1.test.dom 172.30.20.1 FastEthernet1/1 FastEthernet2/0
 neighbor R2.test.dom 172.30.30.1 FastEthernet1/0 FastEthernet1/1
 neighbor R4.test.dom 172.30.40.2 FastEthernet2/0 FastEthernet1/0
 neighbor R5.test.dom 172.30.50.2 FastEthernet2/1 FastEthernet1/0
devices topology layer2 nodes cis4/172.30.99.4
 device-global-id R4.test.dom
 neighbor R3.test.dom 172.30.40.1 FastEthernet1/0 FastEthernet2/0
 neighbor R5.test.dom 172.30.60.2 FastEthernet1/1 FastEthernet1/1
devices topology layer2 nodes cis5/172.30.99.5
 device-global-id R5.test.dom
 neighbor R3.test.dom 172.30.50.3 FastEthernet1/0 FastEthernet2/1
 neighbor R4.test.dom 172.30.60.1 FastEthernet1/1 FastEthernet1/1
[ok][2013-05-06 11:13:10]
```

Listing 19: L2 topology discovery results

Figure 14 and Figure 15 represent the visualization of the L3 and L2 topologies (respectively). The L3 visualization shows the devices and the networks these devices are connected to, while the L2 visualization shows the physical interconnections of the devices. The actual names, as they are configured in NCS, are used for the devices, thus, they are different from the names shown in Figure 13 and the actual names that are configured on the devices (domain names). Note that at the current stage of the project visualization package has some limitations in representation of the actual data structure for topology discovery (see Listings 18 and 19), e.g. it does not fully support multiple links between devices.

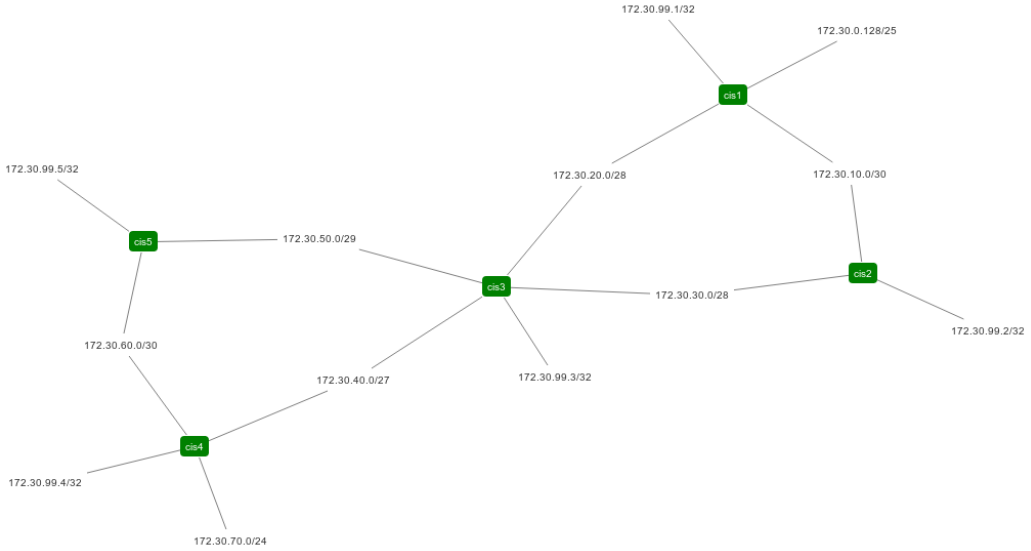

Figure 14: Discovered L3 topology visualization

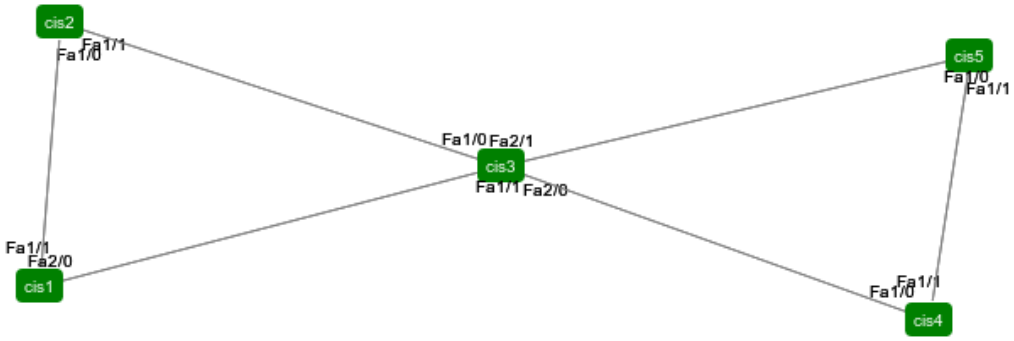

Figure 15: Discovered L2 topology visualization

It may happen that one or more network devices are missing, not supported, or incorrectly configured in NCS. Figure 16 and Figure 17 show an example of a visualized network topology when `cis3` device is improperly configured. Due to the fact the L3 topology discovery "connects" a device to the discovered set of subnets relying on its interfaces' configuration information, it is impossible to recover the part of the topology which includes the missing device. At a minimum we try to utilize the device's management IP address to connect it to one of the discovered subnets (note that this is not the case in Figure 16 as all of the devices in the test network are managed via their loopback interface addresses).

In contrast, CDP provides explicit information for L2 topology discovery as the same information about each link can be retrieved from each of the two devices connected via a link. Therefore, it is possible to recover part of the topology with the missing device; however, there is no way to match this device with an improperly configured device in the NCS configuration.

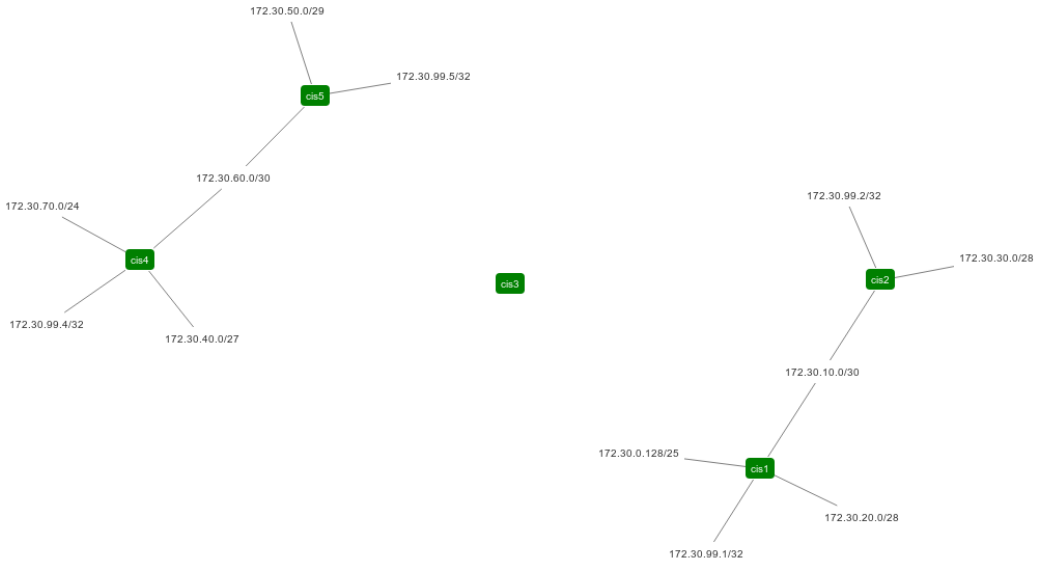

Figure 16: Discovered L3 topology visualization with misconfigured device (cis3)

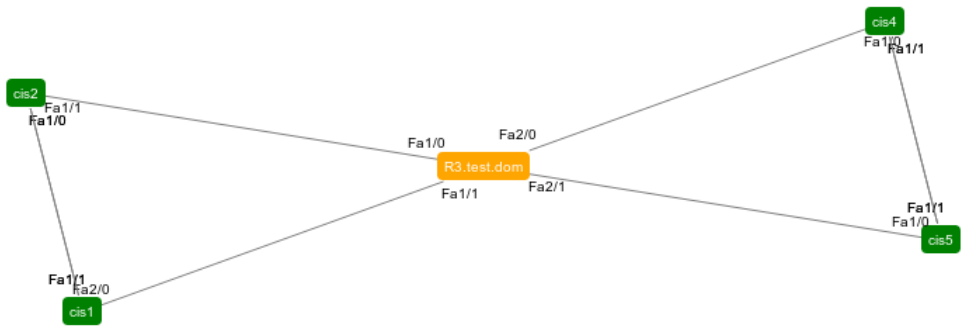

Figure 17: Discovered L2 topology visualization includes the misconfigured device

Having the topology discovery task split into a logical topology discovery and a physical topology discovery allows more fine-grained control over the level of detail of the network map. Additionally, it allows to simplify requirements

for the low-detailed L3 network map while putting greater requirements for the highly-detailed L2 map (as generally the task of building a highly-detailed map requires using more of the device capabilities). This approach seems natural as L3 topology allows the operator to see the map of the core network while the L2 topology provides detailed views of different parts of what is probably an access network.

Since NCS does not yet include native data models for retrieving the required operational data about IP addresses and link-layer connections from a device over NETCONF or SSH (CLI), it is difficult to cover a wide range of different devices which only support other protocols. While retrieving the data only via SNMP and relying only on CDP may not seem a wide variety, including LLDP and NETCONF RPCs would add to the diversity of data sources. Moreover, there is an advantage in using NCS's internal mechanisms for communication with the devices and once there are data models for retrieving operational data via additional protocols (such as NETCONF) then it will be possible to support a greater variety of devices.

Another problem is the interoperability of the protocols for L2 discovery. For example, in the case of CDP and LLDP it would be difficult to produce a topology if some of the devices in the same network only use LLDP and others only use CDP. The physical topology in such a case would generate two distinct maps reflecting the data collected by the two protocols. However, although a network may include different vendors' networking equipment, it is unlikely that a mixture of different equipment would be used in the same part of the network. For this reason we do not expect that there will be a problem generating these two different network maps. It might be possible in the future to use L3 information to connect these two different network maps.

Network discovery, and especially network topology discovery, is a widely discussed topic and there is a lot of work that has been done in this area (see Chapter 2), including developing both theoretical approaches and practical implementations of discovery tools. The topology discovery package developed during this master's thesis project also falls in the range of this type of tools, however, it is difficult to compare this package to other tools, as it is specifically designed as a component for NCS (implying the choice of devices that are to be discovered as well as the method of communication with those devices). Nevertheless, the methods used for topology discovery can be compared.

The general purpose topology discovery tools usually include a device discovery functionality, since they need a starting point which is usually a range of IP addresses or a single IP address. A popular way to proceed once the range has been defined is to perform a so called "ping sweep" which will identify which

devices are online, and usually a traceroute which will provide initial mapping of distance (i.e. number of hops) to a certain device. **Argus** [63] is an example of a tool that utilizes this approach. After the initial mapping is done and the devices to work with are found additional protocols may be used to obtain information about the device interconnections. These protocols are mainly SNMP and link-layer neighbor discovery protocols such as LLDP. The topology discovery package developed for NCS does not include any device discovery component as it works with the devices that are already configured in NCS (although the device discovery functionality has been implemented in a separate module which is described in Chapter 3, the topology discovery module does not depend on this functionality). Argus relies on SNMP only, which may be a limitation with the devices that support a link-layer discovery protocol, but do not have an SNMP support. There are tools that rely on LLDP protocol exclusively; an example of such a tool is an LLDP-walk [64] demonstration, which connects to devices via SSH (CLI) and retrieves LLDP information. A distinctive feature of our topology discovery implementation is its architecture which allows for easy integration with other protocols, thus making it possible to have combinations of the protocols that provide required information and the protocols that are used to retrieve this information.

The following chapter concludes the report. It reviews the goals that have been achieved, discusses the ethical aspects of this work, and suggests some future work that could be done in this area.

Chapter 5

Conclusions and Future work

This chapter concludes the thesis. The following sections give a general summary of the work; discuss the initial goals that were defined for the project and compare them with the outcome of this work. The chapter also discusses some ethical aspects related to this project, and the future work that could built upon the results of this project.

5.1 Project summary and results

This master's thesis project was done at Tail-f Systems. The result of this project is two optional NCS packages that provide network device discovery and network topology discovery functionality. This result reflects the goals originally defined for this project.

The device discovery package provides network device discovery of devices in the specified IP address range, while also determining the type (software and hardware platform) of the devices whenever possible and collecting all the additional parameters required to add the discovered device to NCS's managed devices tree. This package minimizes the effort required by an NCS operator by allowing the operator to automatically save the discovered device(s) into the NCS configuration database together with the relevant parameters and provides an option to add multiple devices with a single command. The device discovery package meets the requirements initially set for the package. The package includes both requested Nmap-based device discovery and a stand-alone device discovery engine which utilizes a credential based approach to device discovery and is better suited for use by an NCS operator. The package's data model is

defined in YANG and the structure of the package meets the requirements for an NCS package.

The topology discovery package provides the functionality to discover a network's topology for those devices configured in NCS. The topology discovery package provides two actions for topology discovery: L3 topology discovery which aims at discovering the logical structure of the network and L2 topology discovery which aims at discovering physical (link-layer) interconnection between devices. The YANG data model defined for the package provides a data structure which permits an extensive representation of the discovered topology. This representation is used as input for an experimental visualization prototype developed for the topology discovery package. The visualization provides a graphical representation of the discovered topology. In the current version of the topology discovery package L2 topology discovery is based on the CDP protocol to collect information about link-layer interconnections between the devices.

The initially planned LLDP based topology discovery method was not realized. The reason behind this was the limited availability of hardware with LLDP support, particularly the unavailability of Juniper equipment; equipment from this vendor was the primary target for this method. However, possible approaches to LLDP based discovery were developed. An approach similar to CDP based discovery may be utilized for LLDP based discovery, when the data is retrieved by NCS using SNMP. Additionally, a NETCONF RPC based approach may be utilized. An advantage of this particular approach is that it eliminates the need for an additional protocol (SNMP in our case), but requires devices to support NETCONF and requires developing data models which reflect the RPC structure. While developing data models is a demanding task, it is not the primary concern of this approach, but rather the need for the support of NETCONF on different devices. As Juniper devices natively support the NETCONF protocol they were considered the primary target for LLDP based discovery.

Although LLDP based topology discovery was not implemented, the architecture of the package allows the package to easily be extended to support LLDP. The package can be also extended to support other protocols for device communication to cover an even wider range of different types of devices. The prerequisite for including this functionality into the package is the presence of the relevant data models in NCS.

The following section describes the ethical considerations behind this work. The section also discusses the authors' view of the intended use of software which was developed and the use of the material presented in this thesis.

5.2 Ethical considerations

Network scanning is a highly debated topic from an ethical perspective and especially from a legal perspective. This thesis covers some aspects and methodologies used for network scanning and discovery, additionally the software developed during this project provides network scanning functionality. We, as the authors of this work, highly oppose any unintended and malicious usage of the results of this work. We expect that the software developed during this project would be used only by network administrators in their own networks, or used with the permission of the network's administrator. The content of this thesis is intended to provide research based insight into network scanning and should be useful for other research, educational, or other legitimate purposes.

While developing the discovery components for NCS we assumed that these components would only be used by the respective network's operators or otherwise persons responsible for a network. While the Nmap-based discovery component relies on Nmap to provide its functionality, all the legal considerations applied to Nmap should be applied to the usage of this component as well. Moreover, the stand-alone discovery engine was not developed as a general purpose network scanner as it utilizes credential(s) based discovery as its main feature in order to provide precise and accurate information about the devices that are discovered. The operator must specify the credentials to be used during the discovery process. While this requires more effort from the operator, it is less effort that would be required to manually configure each device. A password guessing feature could somewhat simplify the process, however, this seems contrary to the ethical norms of such a discovery component for NCS. The credentials based discovery somewhat resembles a dictionary approach to password guessing, when an operator defines a long list of credentials, as credentials are sequentially used to access each of the devices. Although, the module might be utilized as a tool in conjunction with a brute-force password guessing attack, we would highly discourage potential abusers from doing so. Additionally, the discovery component does not provide any optimizations (such as timeouts, IDS/IPS evasion, etc.) for this task, as we have assumed that the operator who runs this package has all the required authorizations and that this operator is authorized to use this package.

The usage of components is controlled by NCS's access control and security mechanisms; the components themselves do not implement any additional usage restrictions. Thus, in multiuser NCS environments, the NCS administrator should configure suitable usage policies for these new components. The only security feature implemented for these new components is the encryption of the credentials

stored in the configuration database and the valid credentials that worked on a specific device when stored into the operational database. Encrypting strings is a built-in feature of NCS, which is controlled by the NCS configuration. The NCS administrator can specify the encryption algorithm to be used and the associated parameters (encryption keys, initialization vectors, etc.). The NCS administrator is advised to select an appropriate encryption algorithm and associated parameters, while adhering to the general practice of securing access to the encryption keys.

The following section describes some potential future work related to this project. It also discusses some possible future work in the area of network discovery from the perspective of this thesis project.

5.3 Future work

The device and topology discovery packages provide the required discovery functionality - as this functionality was defined within the scope of this thesis project. However, these packages are still an experimental feature and do not provide the functionality which might be ultimately required. Future work should make these new discovery components more complete in order to provide a feature rich network discovery solution to Tail-f Systems NCS customers. The features described below would make these new discovery components more complete:

- Adding IPv6 support to the stand-alone discovery engine. This would make the device discovery package compliant with the rapidly changing industry's demands and the increasing spread of IPv6.
- Enabling support for device communication via NETCONF while performing device discovery.
- Adding LLDP based physical discovery to the topology discovery component. This feature would allow the topology discovery component to cover additional device types. For Juniper equipment, it would be good to implement the NETCONF RPC based approach in order to retrieve additional operational data, and laying the ground for supporting future NETCONF enabled devices. It might also be desirable to implement Cisco's CLI based approach to support those Cisco devices which do not run an SNMP agent.

- Adding support for sequential neighbor discovery to the topology discovery module. This feature will help to identify modifications made to the network without a need to query all the devices again, but starting from a particular device in the network and traversing the neighbors connected to that device.
- The experimental visualization of discovered topology representations should be incorporated into the NCS WebUI. This will facilitate a better overall representation of the discovered data, as well as possibly linking the visualization elements to the NCS WebUI objects (for instance when a device is presented on the visualized map it could be a reference to the device actually configured in NCS), thus allowing for richer functionality of the visual topology representation.

In general it would be interesting to follow the development of NETCONF, including the development of the YANG language. As more devices become NETCONF enabled it should be much easier for NCS to support them. The development that is being done in this area includes enhancements of the NETCONF protocol and enabling this protocol support on different devices.

It may be beneficial to follow the development of the various graph visualization libraries. A particularly interesting development would be a library aimed specifically at generating network topology maps.

The development of open network equipment, such as proposed by the Open Compute Project [65], and the increasing use of OpenFlow [66] switching both increases the variety of network equipment that must be managed and reduces the time scale during which management decisions have to be made. A possible future work is to extend the results of this thesis in these areas.

References

[1] "Nmap - Legal Notices." [Online]. Available: http://nmap.org/book/man-legal.html [Accessed: 12-Mar-2013].

[2] J. Case, R. Mundy, D. Partain, and B. Stewart, "Introduction and Applicability Statements for Internet-Standard Management Framework," RFC 3410 (Informational), Internet Engineering Task Force, Dec. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3410.txt

[3] K. McCloghrie, D. Perkins, and J. Schoenwaelder, "Structure of Management Information Version 2 (SMIv2)," RFC 2578 (INTERNET STANDARD), Internet Engineering Task Force, Apr. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2578.txt

[4] R. Stadler, lectures notes for the course EP2300 Management of Networks and Networked Systems, KTH Royal Institute of Technology, Aug.-Oct. 2012, unpublished.

[5] J. Yu and I. Al Ajarmeh, "An Empirical Study of the NETCONF Protocol," in *Sixth International Conference on Networking and Services (ICNS)*, March 2010, pp. 253–258.

[6] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network Configuration Protocol (NETCONF)," RFC 6241 (Proposed Standard), Internet Engineering Task Force, Jun. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6241.txt

[7] J. Schönwälder, M. Björklund, and P. Shafer, "Network configuration management using NETCONF and YANG," in *IEEE Communications Magazine*, vol. 48, no. 9, September 2010, pp. 166–173.

[8] M. Wasserman, "Using the NETCONF Protocol over Secure Shell (SSH)," RFC 6242 (Proposed Standard), Internet Engineering Task Force, Jun. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6242.txt

[9] M. Badra, “NETCONF over Transport Layer Security (TLS),” RFC 5539 (Proposed Standard), Internet Engineering Task Force, May 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5539.txt

[10] T. Goddard, “Using NETCONF over the Simple Object Access Protocol (SOAP),” RFC 4743 (Historic), Internet Engineering Task Force, Dec. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4743.txt

[11] E. Lear and K. Crozier, “Using the NETCONF Protocol over the Blocks Extensible Exchange Protocol (BEEP),” RFC 4744 (Historic), Internet Engineering Task Force, Dec. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4744.txt

[12] “IETF Network Configuration Working Group.” [Online]. Available: http://datatracker.ietf.org/wg/netconf/charter/ [Accessed: 5-Feb-2013].

[13] M. Bjorklund, “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF),” RFC 6020 (Proposed Standard), Internet Engineering Task Force, Oct. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc6020.txt

[14] Tail-f Systems, “NCS User Guide,” Dec. 2012, unpublished.

[15] J. Schönwälder and H. Langendörfer, “How to Keep Track of Your Network Configuration,” in *Proceedings of the 7th USENIX conference on System administration (LISA 1993)*, 1993, pp. 189–193.

[16] R. Braden, “Requirements for Internet Hosts - Communication Layers,” RFC 1122 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1989, updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt

[17] Hwa-Chun Lin, Shou-Chuan Lai, and Ping-Wen Chen, “An algorithm for automatic topology discovery of IP networks,” in *IEEE International Conference on Communications, ICC 1998*, vol. 2, 1998, pp. 1192–1196.

[18] W. Liu, “Research on Remote Operating System Detection Using Libnet,” in *International Conference on Industrial and Information Systems, IIS 2009*, 2009, pp. 101–103.

[19] Jiang Wei-hua, Li Wei-hua, and Du Jun, “The application of ICMP protocol in network scanning,” in *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2003*, 2003, pp. 904–906.

[20] “Nmap Security Scanner.” [Online]. Available: http://nmap.org [Accessed: 5-Feb-2013].

[21] P. Dobrev, et al., “Device and service discovery in home networks with OSGi,” *Communications Magazine, IEEE*, vol. 40, no. 8, pp. 86–92, 2002.

[22] A. Wils, et al., “Device discovery via residential gateways,” *International Conference on Consumer Electronics, 2002*, vol. 48, no. 3, pp. 478–483, 2002.

[23] G.G. Richard, “Service advertisement and discovery: enabling universal device cooperation,” *Internet Computing, IEEE*, vol. 4, no. 5, pp. 18–26, 2000.

[24] JiaBin Yin et al., “SNMP-based network topology discovery algorithm and implementation,” in *Fuzzy Systems and Knowledge Discovery (FSKD), 2012, 9th International Conference, IEEE*, 2012, pp. 2241–2244.

[25] Han Yan, “The study on network topology discovery algorithm based on SNMP protocol and ICMP protocol,” in *Software Engineering and Service Science (ICSESS), 2012, 3rd International Conference, IEEE*, 2012, pp. 665–668.

[26] J. Postel, “Internet Protocol,” RFC 791 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1349, 2474. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[27] J. Postel, “Transmission Control Protocol,” RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[28] “IP Option Numbers.” [Online]. Available: http://www.iana.org/assignments/ip-parameters/ip-parameters.xml [Accessed: 13-Feb-2013].

[29] K. Ramakrishnan, S. Floyd, and D. Black, “The Addition of Explicit Congestion Notification (ECN) to IP,” RFC 3168 (Proposed Standard), Internet Engineering Task Force, Sep. 2001, updated by RFCs 4301, 6040. [Online]. Available: http://www.ietf.org/rfc/rfc3168.txt

[30] V. Jacobson, R. Braden, and D. Borman, “TCP Extensions for High Performance,” RFC 1323 (Proposed Standard), Internet Engineering Task Force, May 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1323.txt

[31] J.M. Allen, “OS and Application Fingerprinting Techniques.” [Online]. Available: http://www.sans.org/reading_room/whitepapers/authentication/os-application-fingerprinting-techniques_32923 [Accessed: 5-Feb-2013].

[32] G. Lyon, “Nmap network scanning: official Nmap project guide to network discovery and security scanning,” 2011. [Online]. Available: http://nmap.org/book/toc.html

[33] “IEEE std 802-2001, Standard for Local and Metropolitan Area Networks: Overview and Architecture,” February 2002.

[34] “IEEE Public OUI List.” [Online]. Available: http://standards.ieee.org/develop/regauth/oui/oui.txt [Accessed: 14-Feb-2013].

[35] T. Aura, “Cryptographically Generated Addresses (CGA),” RFC 3972 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 4581, 4982. [Online]. Available: http://www.ietf.org/rfc/rfc3972.txt

[36] “Arp Scan.” [Online]. Available: http://www.nta-monitor.com/tools-resources/security-tools/arp-scan [Accessed: 14-Feb-2013].

[37] M. Srinivasan, “Tutorial on the Link Layer Discovery Protocol.” [Online]. Available: http://www.eetimes.com/design/communications-design/4009357/Tutorial-on-the-Link-Layer-Discovery-Protocol/ [Accessed: 13-Feb-2013].

[38] V.Z. Attar and P. Chandwadkar, “Network Discovery Protocol LLDP and LLDP-MED,” in *International Journal of Computer Applications*, vol. 1, no. 9, 2010, pp. 93–97.

[39] Luo Junhai, Fan Mingyu, and Ye Danxia, “Research on Topology Discovery for IPv6 Networks,” in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007, SNPD 2007. 8th ACIS International Conference*, vol. 3, 2007, pp. 804–809.

[40] Shen Zengwei and Zhou Gang, “Research of the topology auto-discovery approach in the ipv6 access network,” in *Computers, Communications, Signal Processing with Special Track on Biomedical Engineering, 2005, CCSP 2005. 1st International Conference*, 2005, pp. 96–100.

[41] S. Deering, “Host extensions for IP multicasting,” RFC 1112 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1989, updated by RFC 2236. [Online]. Available: http://www.ietf.org/rfc/rfc1112.txt

[42] R. Hinden and S. Deering, “IP Version 6 Addressing Architecture,” RFC 4291 (Draft Standard), Internet Engineering Task Force, Feb. 2006, updated by RFCs 5952, 6052. [Online]. Available: http://www.ietf.org/rfc/rfc4291.txt

[43] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, “Neighbor Discovery for IP version 6 (IPv6),” RFC 4861 (Draft Standard), Internet Engineering Task Force, Sep. 2007, updated by RFC 5942. [Online]. Available: http://www.ietf.org/rfc/rfc4861.txt

[44] J. Arkko, J. Kempf, B. Zill, and P. Nikander, “SEcure Neighbor Discovery (SEND),” RFC 3971 (Proposed Standard), Internet Engineering Task Force, Mar. 2005, updated by RFCs 6494, 6495. [Online]. Available: http://www.ietf.org/rfc/rfc3971.txt

[45] S. Thomson, T. Narten, and T. Jinmei, “IPv6 Stateless Address Autoconfiguration,” RFC 4862 (Draft Standard), Internet Engineering Task Force, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4862.txt

[46] Cisco Systems, “Cisco IOS Configuration Fundamentals Command Reference - show protocols through showmon.” [Online]. Available: http://www.cisco.com/en/US/docs/ios/fundamentals/command/reference/cf_s4.html [Accessed: 04-Mar-2013].

[47] Juniper Networks, “show version - Technical Documentation.” [Online]. Available: http://www.juniper.net/techpubs/en_US/junos/topics/reference/command-summary/show-version.html [Accessed: 04-Mar-2013].

[48] “uname(1) - Linux man page.” [Online]. Available: http://linux.die.net/man/1/uname [Accessed: 04-Mar-2013].

[49] R. Presuhn, “Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP),” RFC 3416 (INTERNET STANDARD), Internet Engineering Task Force, Dec. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3416.txt

[50] J. Case, M. Fedor, M. Schoffstall, and J. Davin, “Simple Network Management Protocol (SNMP),” RFC 1157 (Historic), Internet Engineering Task Force, May 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt

[51] U. Blumenthal and B. Wijnen, “User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3),” RFC 3414 (INTERNET STANDARD), Internet Engineering Task Force, Dec.

2002, updated by RFC 5590. [Online]. Available: http://www.ietf.org/rfc/rfc3414.txt

[52] "SNMP4J - The Object Oriented SNMP API for Java Managers and Agents." [Online]. Available: http://www.snmp4j.org/ [Accessed: 11-Mar-2013].

[53] J. Postel, "Internet Control Message Protocol," RFC 792 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 950, 4884, 6633. [Online]. Available: http://www.ietf.org/rfc/rfc792.txt

[54] C. Wikström, Tail-f Systems, Jan. 2013, private communication.

[55] K. McCloghrie and F. Kastenholz, "The Interfaces Group MIB," RFC 2863 (Draft Standard), Internet Engineering Task Force, Jun. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2863.txt

[56] S. Routhier, "Management Information Base for the Internet Protocol (IP)," RFC 4293 (Proposed Standard), Internet Engineering Task Force, Apr. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4293.txt

[57] "Cisco CDP MIB." [Online]. Available: http://tools.cisco.com/Support/SNMP/do/BrowseMIB.do?local=en&step=2&mibName=CISCO-CDP-MIB [Accessed: 3-May-2013].

[58] C. Swinehart, "Arbor.js: a graph visualization library using web workers and jQuery." [Online]. Available: http://arborjs.org/ [Accessed: 3-May-2013].

[59] "Graphviz - Graph Visualization Software." [Online]. Available: http://www.graphviz.org/ [Accessed: 15-May-2013].

[60] "The DOT Language." [Online]. Available: http://www.graphviz.org/doc/info/lang.html [Accessed: 15-May-2013].

[61] "Canviz - JavaScript library for drawing Graphviz graphs to a web browser canvas." [Online]. Available: http://code.google.com/p/canviz/ [Accessed: 15-May-2013].

[62] "RadialNet." [Online]. Available: http://www.dca.ufrn.br/~joaomedeiros/radialnet/ [Accessed: 15-May-2013].

[63] "Project Argus - Network topology discovery, monitoring, history, and visualization." [Online]. Available: http://www.cs.cornell.edu/boom/1999sp/projects/networktopology/topology.html [Accessed: 17-May-2013].

[64] J. Schulman, “Simple demonstration of "walking LLDP" in a Junos network to create a connection map.” [Online]. Available: https://gist.github.com/jeremyschulman/4546586 [Accessed: 17-May-2013].

[65] “Open Compute Project.” [Online]. Available: http://www.opencompute.org/ [Accessed: 15-May-2013].

[66] “OpenFlow.” [Online]. Available: http://www.openflow.org/ [Accessed: 15-May-2013].

Appendix A

Device discovery data model

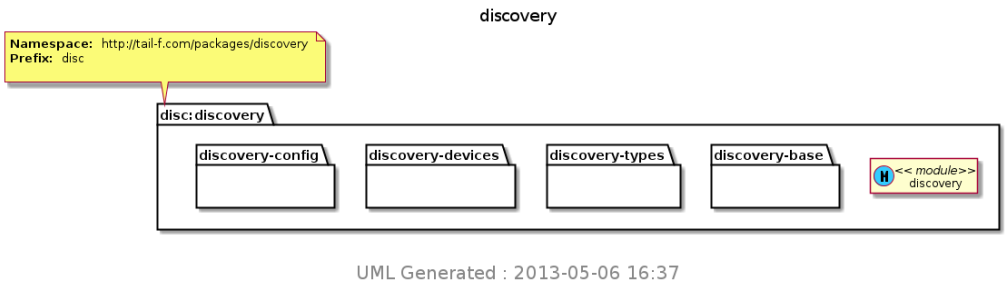

Figure 18: Discovery data model: discovery.yang

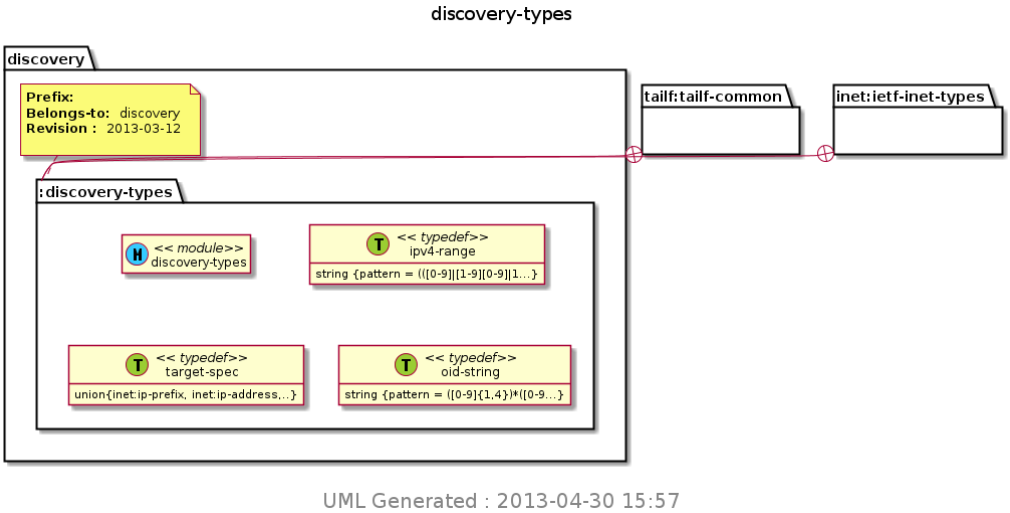

Figure 19: Discovery data model: discovery-types.yang

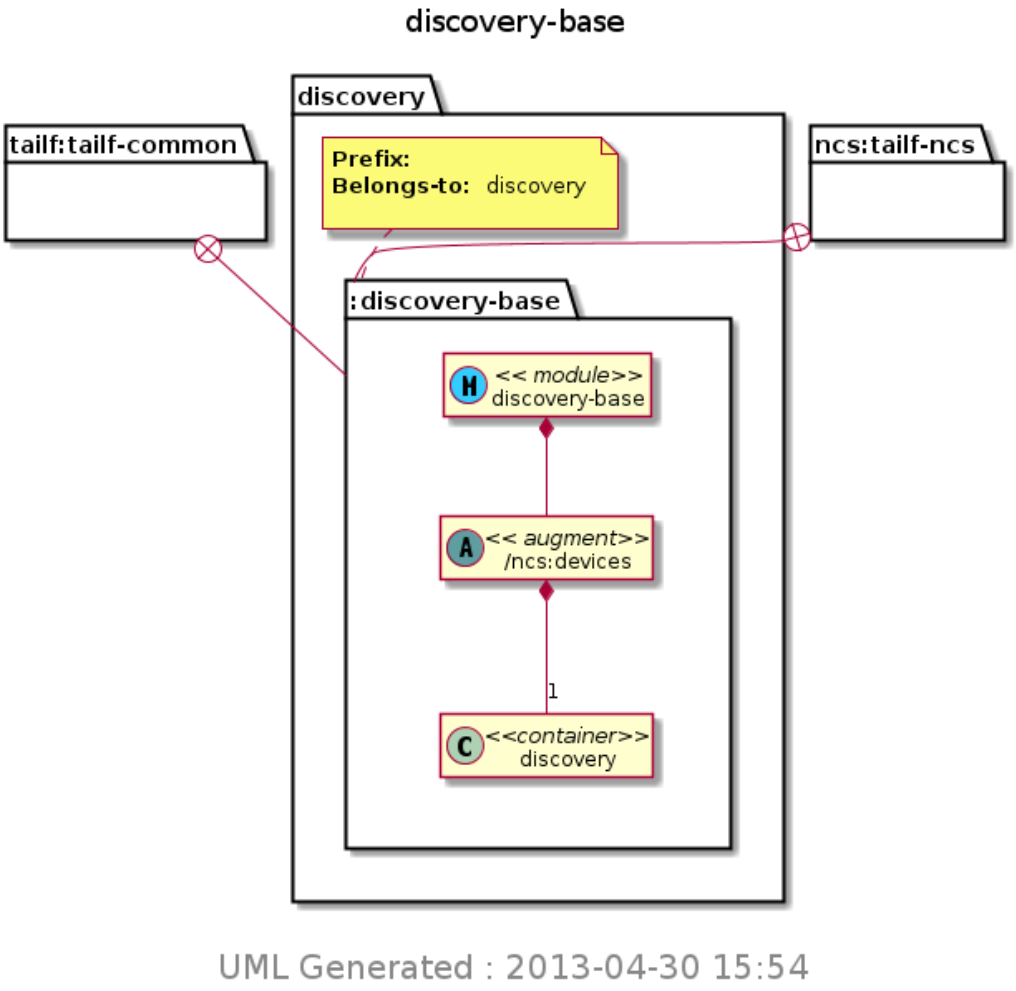

Figure 20: Discovery data model: discovery-base.yang

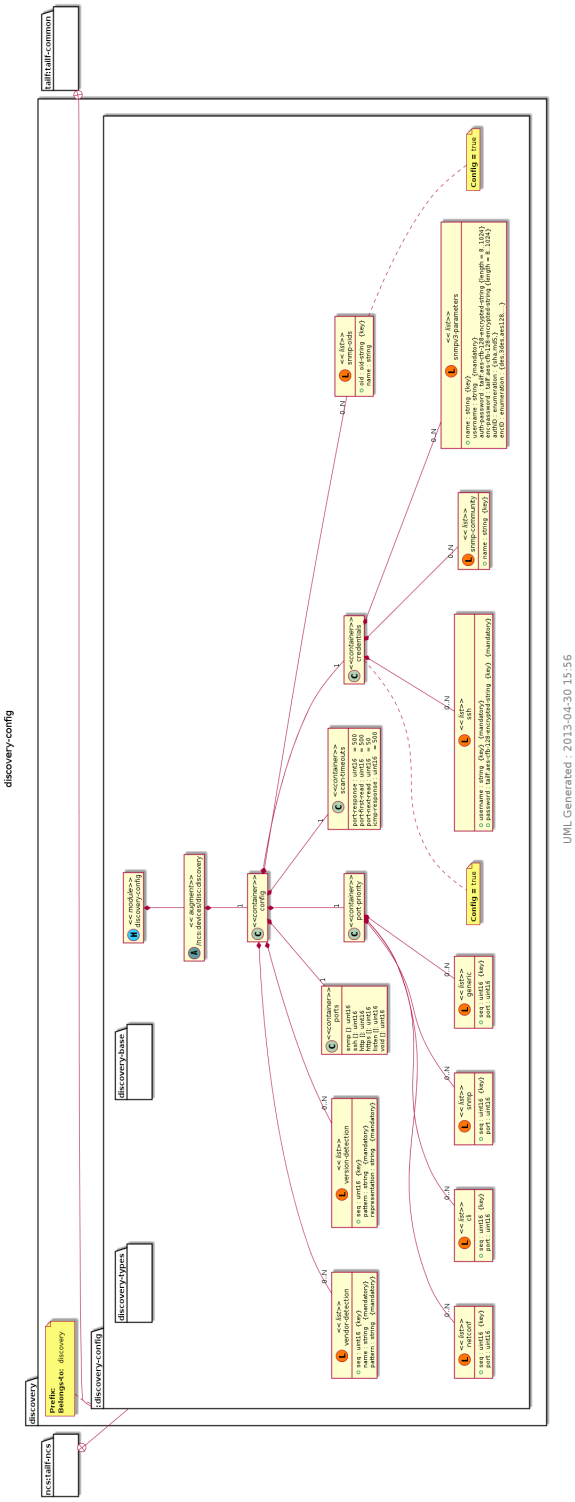

Figure 21: Discovery data model: discovery-config.yang

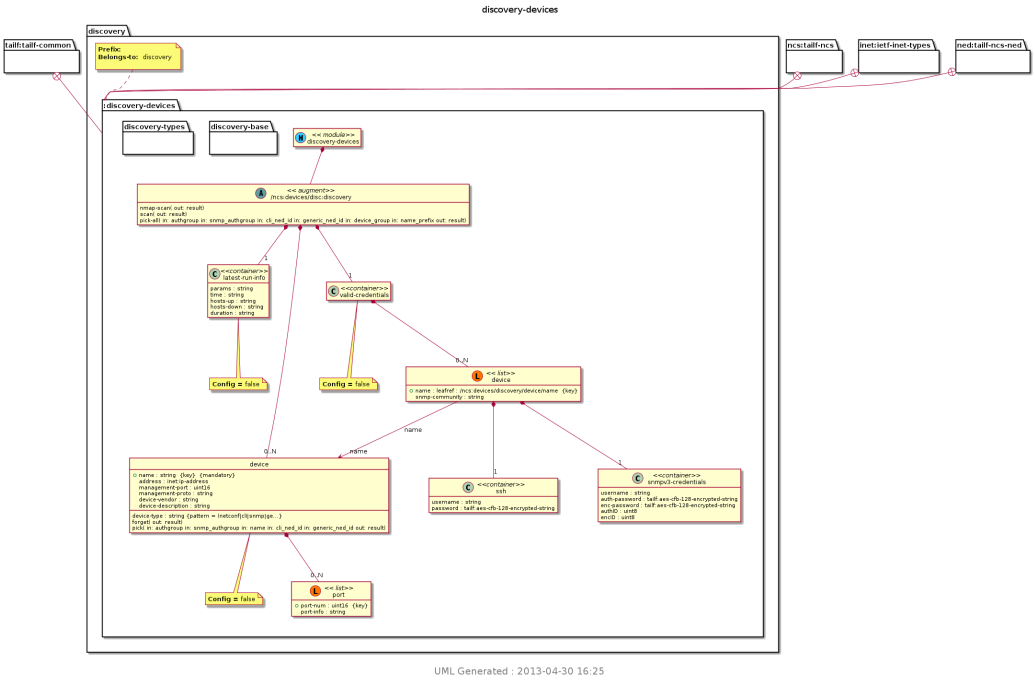

Figure 22: Discovery data model: discovery-devices.yang

Appendix B

Device discovery architecture

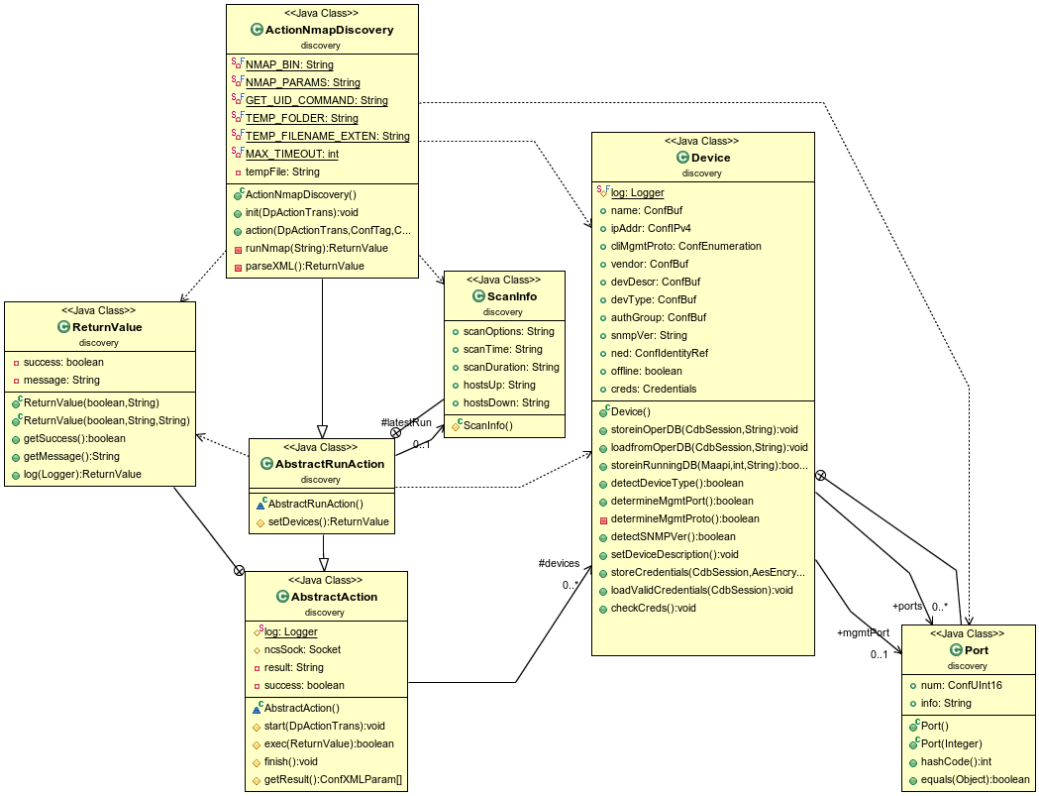

Figure 23: Discovery component: Nmap-based discovery

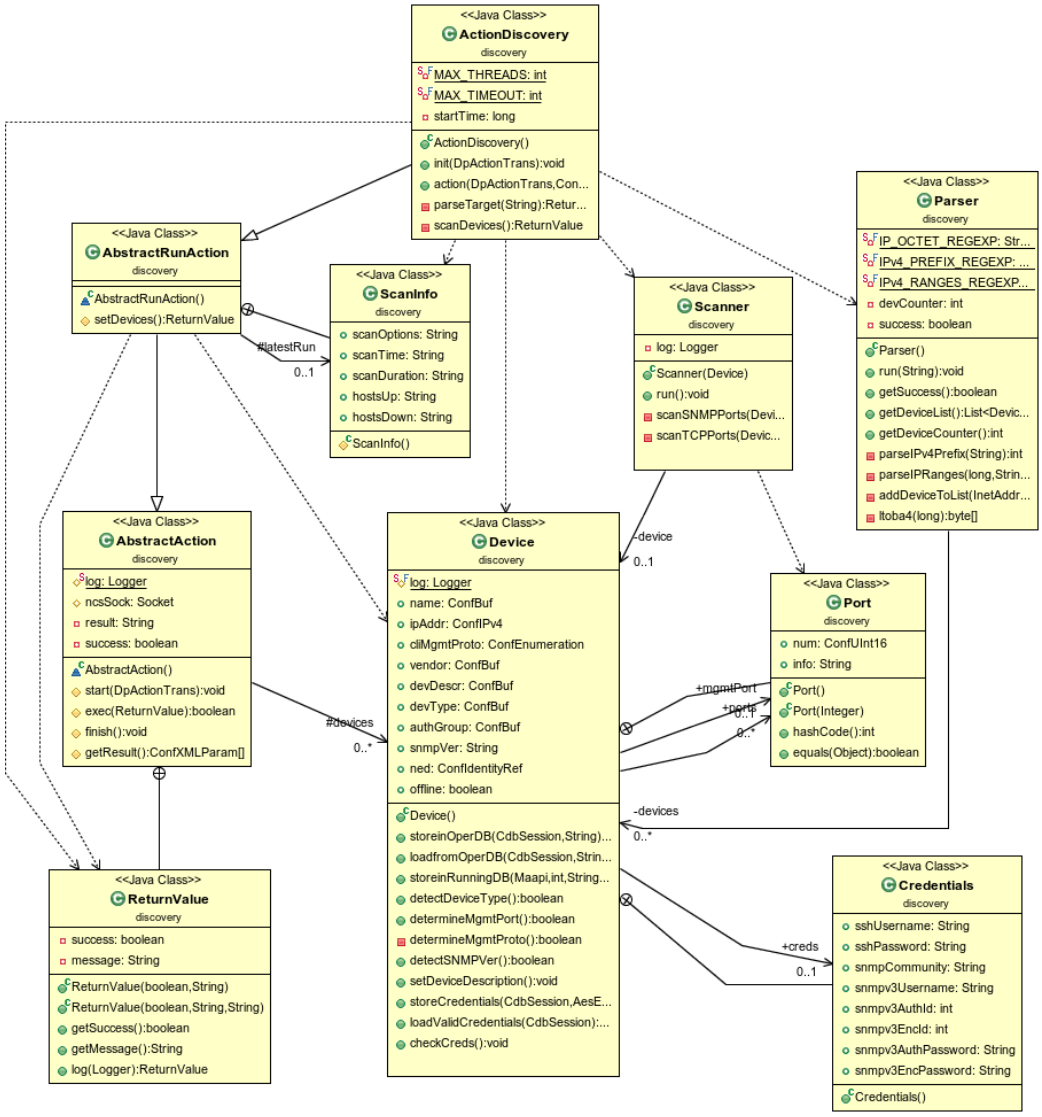

Figure 24: Discovery component: Stand-alone discovery engine

Figure 25: Discovery component: loading devices into NCS

Appendix C

Topology discovery data model

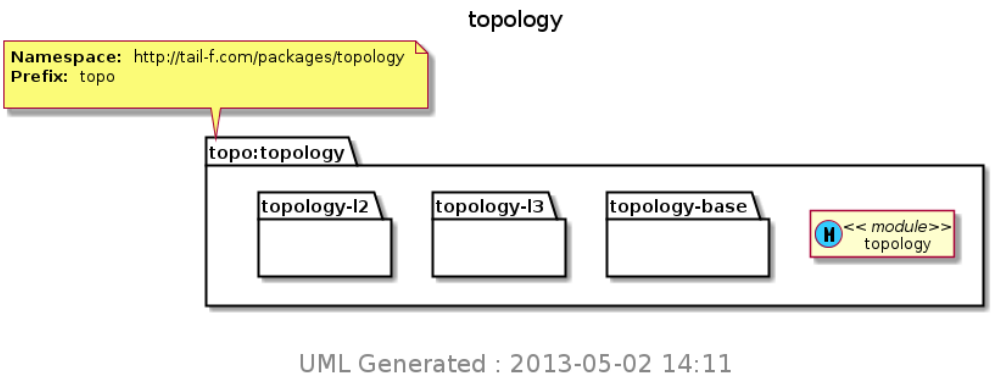

Figure 26: Topology data model: topology.yang

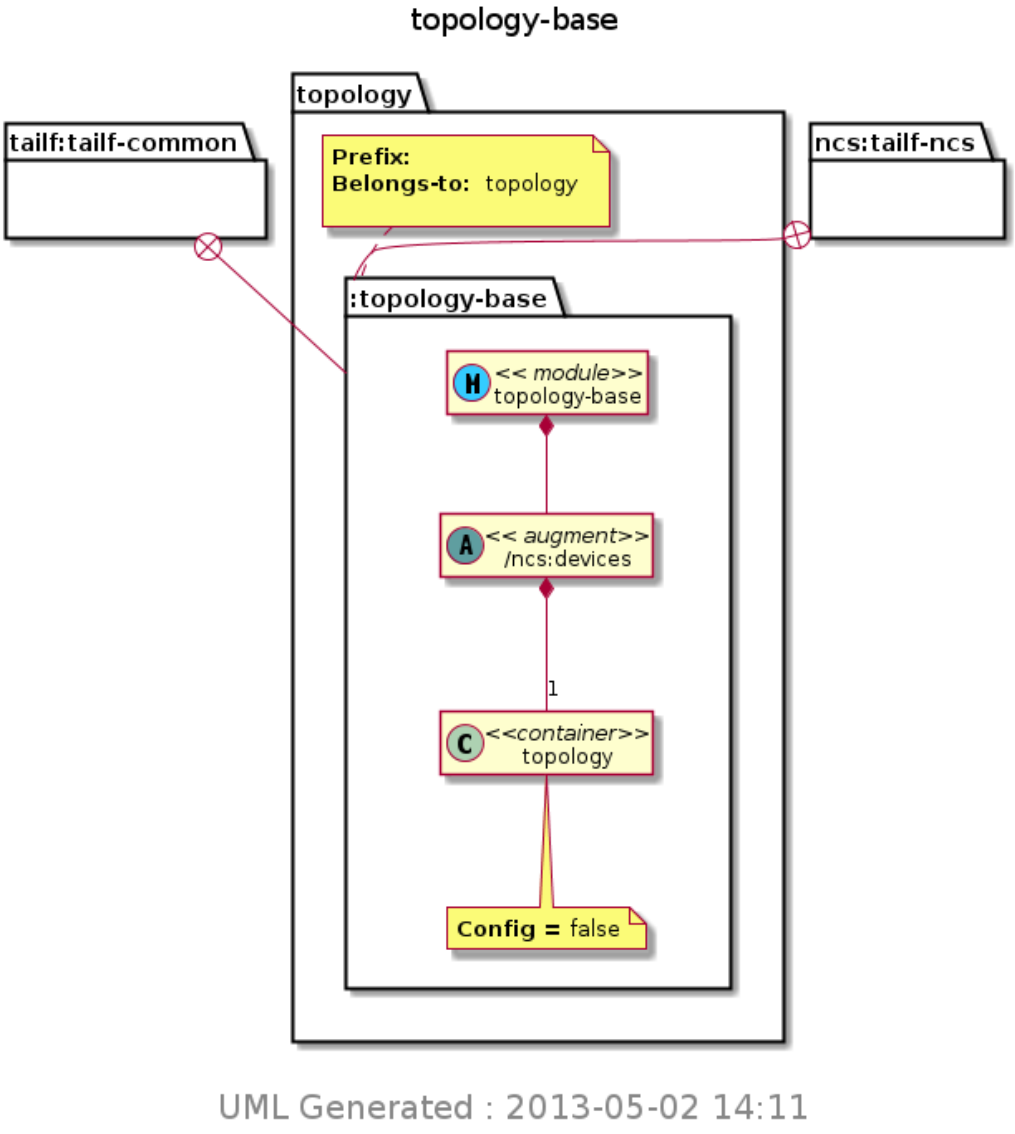

Figure 27: Topology data model: topology-base.yang

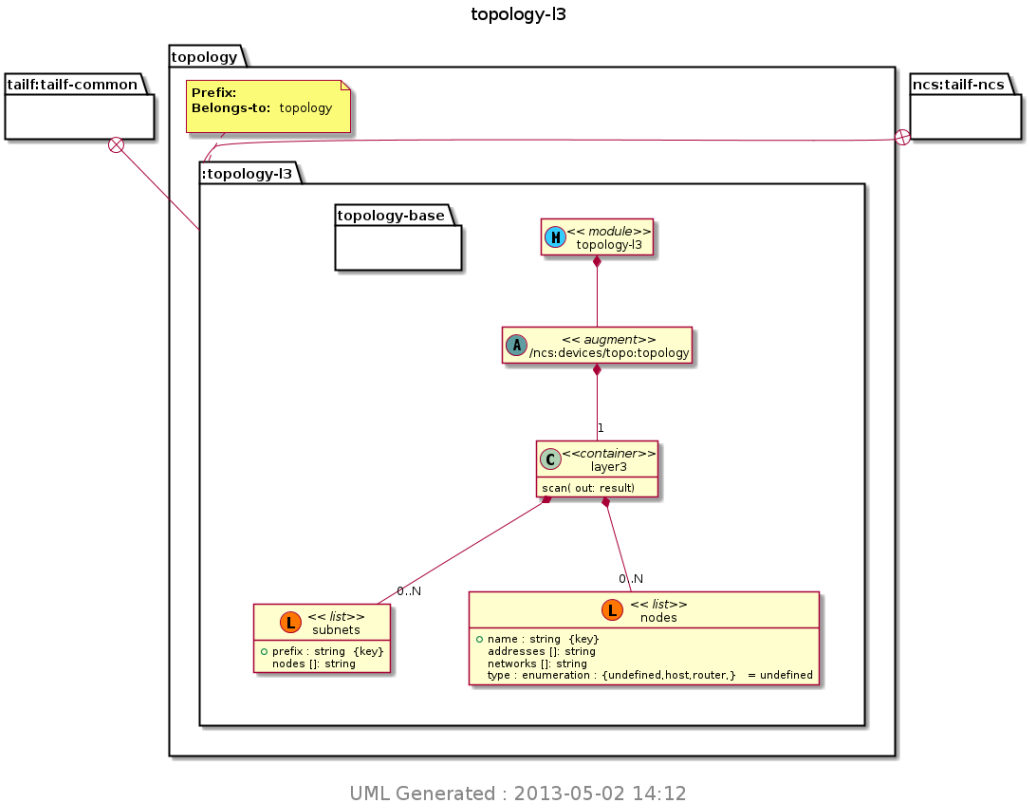

Figure 28: Topology data model: topology-l3.yang

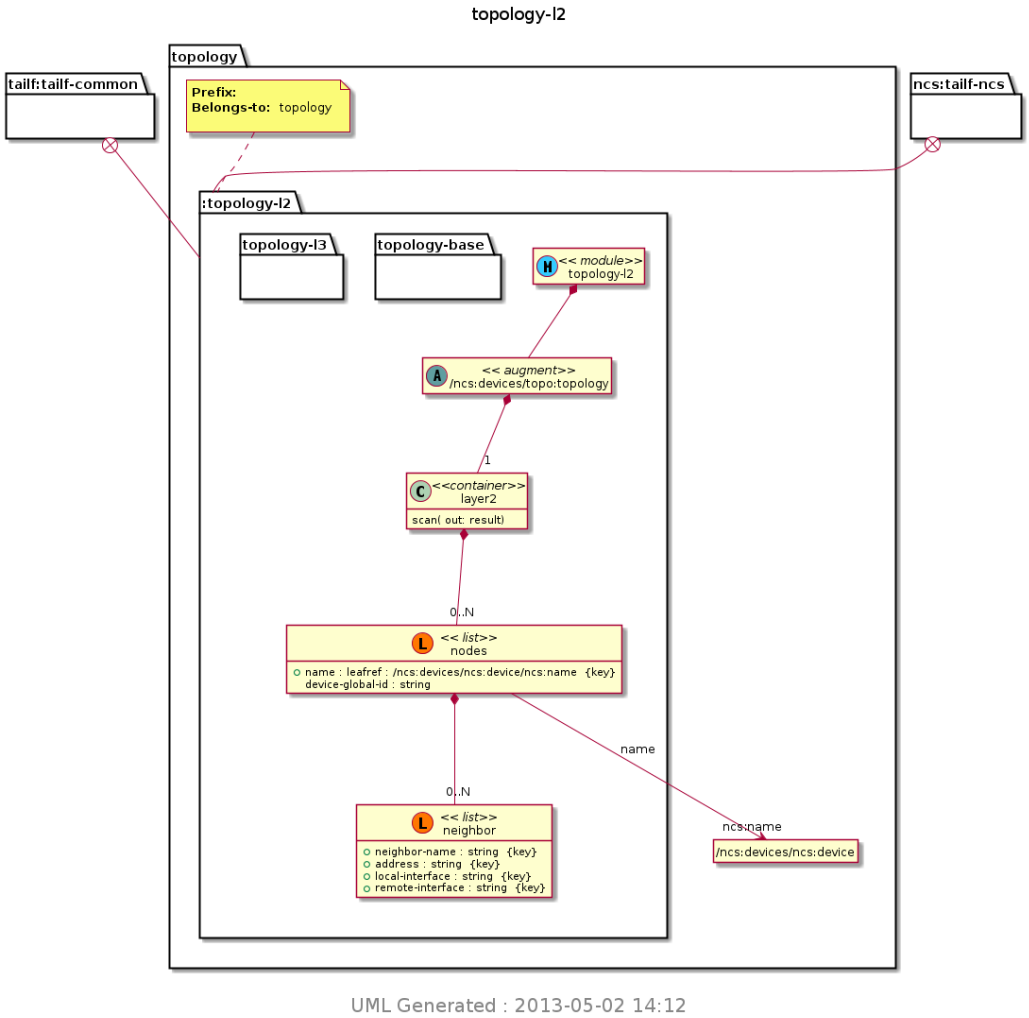

Figure 29: Topology data model: topology-l2.yang

Appendix D

Topology discovery architecture

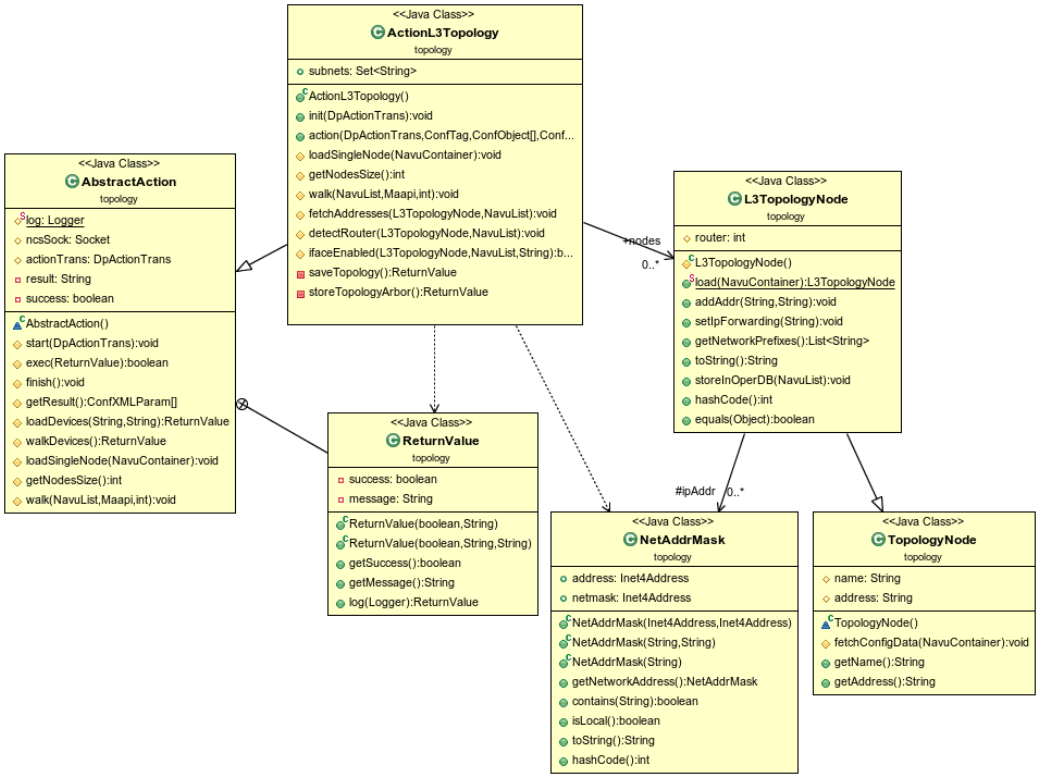

Figure 30: Topology component: L3 topology discovery

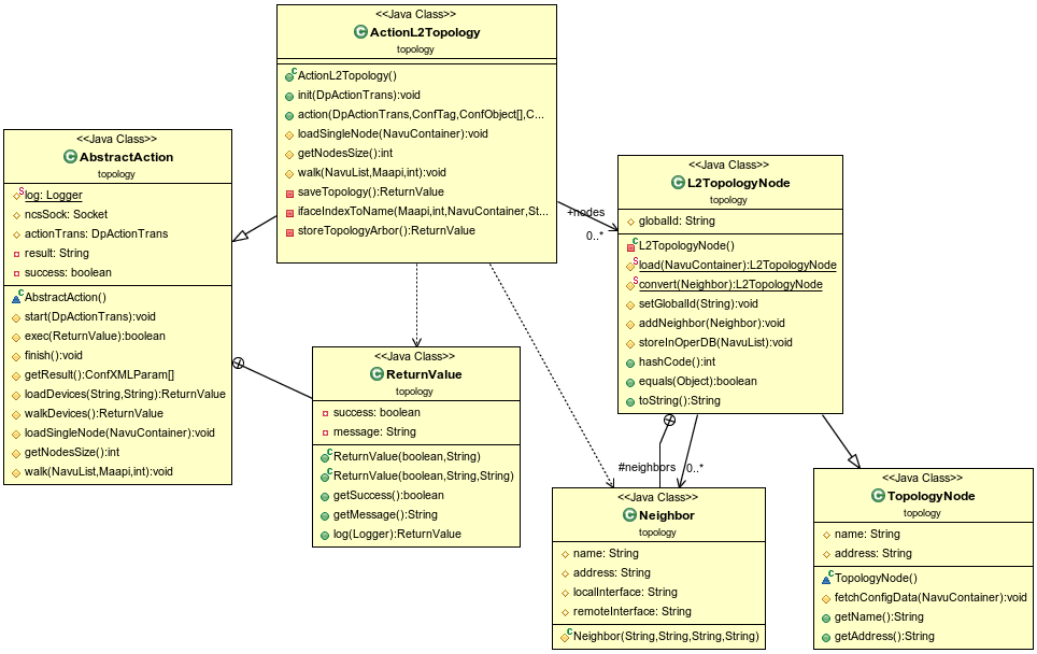

Figure 31: Topology component: L2 topology discovery

TRITA-ICT-EX-2013:90

www.kth.se